Denotational Proof Languages

Kostas Arkoudas

November 24, 2000

"Language shapes the way we think, and determines what we can think about."

B. L. Whorf

Abstract

This dissertation introduces *denotational proof languages* (DPLs). DPLs can be used to express and check (validate) formal proofs. DPL proofs are typically readable, writable, compact, and easy to check. Syntactically, DPL proofs are generated by context-free grammars in such a way that the text of the proof reflects its logical structure. Semantically, the formal meaning of a DPL proof is specified relative to a given *assumption base*, which is a set of premises, i.e., a set of propositions that we take for granted for the purposes of the proof. The key idea is that if a DPL proof is sound, then its meaning (denotation) is the conclusion established by the proof; if the proof is unsound, then its meaning is *error*. To obtain that meaning, we *evaluate* the proof in accordance with the formal semantics of the language. Evaluation will either produce the advertised conclusion, which will verify that the proof is sound, or else it will generate an error, which will indicate that the proof is unsound. Thus in DPLs evaluation is tantamount to proof checking.

We illustrate the core ideas by introducing $\mathcal{CND}$, a simple DPL for classical first-order logic with equality. We develop the theory of $\mathcal{CND}$ in detail, prove its soundness and completeness, present dozens of sample proofs, and develop a number of proof optimization procedures that remove redundancies and unnecessary "detours". We also discuss $\mathcal{CND}$ as a formal analysis of the concept of deduction, and compare it with previous formalizations such as Hilbert calculi, sequent systems, and proof trees. We then introduce the $\lambda\mu$-calculus, a terse formal system that can be viewed as a unifying framework for DPLs, much as the λ-calculus can be seen as a unifying formal framework for functional programming languages. The $\lambda\mu$-calculus allows for proof abstraction via *methods*. Methods are to proofs what procedures are to computations. They are essential for task decomposition and complexity management in proof engineering. Methods are made possible by a careful integration of computation and deduction: the $\lambda\mu$-calculus can be used both for programming and for proofs. More interestingly, we show how methods, when combined with recursion, pattern matching, etc., can be used for proof search and theorem proving. We work out the rudiments of the theory and metatheory of the $\lambda\mu$-calculus, and present several examples of $\lambda\mu$ systems.

Contents

"These things are beyond all use,
And I do fear them."
William Shakespeare, Julius Caesar

1 Background

1.1 Motivation and history

The original motivation for this work was practical: it came from software engineering. Olin Shivers and I had been thinking for a while about a programming environment in which the programmer could make and prove local assertions about this or that piece of code: that a certain parameter can be allocated on the stack, that an array index is never out of bounds, that a function is tail-recursive, or even more involved properties, say that a function is associative. Statements of this sort, along with their proofs, would clearly constitute valuable information about various aspects of the program's run-time behavior. And if these properties were desired to be invariant, then every time the code changed the proofs would have to be accordingly modified to ensure that the assertions were still valid.

Besides serving as good formal documentation, this programmer-provided information could also be used by a smart compiler for optimization purposes (allocate this variable on the stack rather than on the heap; leave out array-bounds checking code from this loop; and so on). Note that we were not concerned with program verification, i.e., with proving that a program is totally correct with respect to a given specification, a task that we regarded as too ambitious to be practical. We were only envisioning an environment in which the programmer could supply the compiler with little pieces of information about certain aspects of its behavior, along with *proofs* to guarantee that the supplied information was correct.

For this to be possible, of course, the compiler must be able to understand the supplied information and to check that the proofs are sound. More specifically, the programmer-compiler dialogue must transpire in a mutually understood formal vocabulary; the proofs must be written in a language that the compiler can interpret; and the assertions must follow from certain axioms postulated by the compiler, for the compiler can only accept the logical consequences of some premises if it can accept the premises themselves. In short, the compiler (or some other trusted agent) must publicize a theory that axiomatizes whatever aspects of the programming language are of interest. The programmers would then be able to make and prove formal statements in that theory. Thus we come to the subject of what we have dubbed *proof engineering*: working with formal theories, and, in particular, developing, checking, and maintaining formal proofs.

Developing a research language and compiler was not a challenge, but not so for the formal framework in which to set up the necessary infrastructure. In what formal notation would the theories and the proofs be expressed? First-order logic, modal logic, higher-order logic, a quantifier-free equational logic in the style of Boyer-Moore? There were many options available, and a number of real-world implementations of systems that could be used: HOL, LF, Isabelle, PVS, Boyer-Moore, just to mention a few. Thus I decided to go about familiarizing myself with these systems in order to arrive at an educated decision. I had a fairly strong background in logic, so I wouldn't have to spend any time actually learning about logic and theories and so on; it was just a matter of reading up on the user manuals.

After a few months of research I felt that I had reached an impasse. On one hand, type-based systems of the Automath variety, although remarkably ingenious in many respects, seemed to require a lot of machinery—such as higher-order abstract syntax, dependent types, kinds, etc.—that is specific to that methodology and peripheral to the usual concerns of the proof engineer. Say that an engineer has a proof that a given circuit performs binary addition correctly, and simply wants a good formal notation in which to express and check that proof. It is not clear why he[1] should have to master higher-order abstract syntax, dependent types, etc., for such a simple task. In fact it is not even clear that he *could* master all that machinery, realistically speaking; the concepts in question are far from easy, and their solid understanding demands a significant effort—a substantially greater effort than, say, learning a new programming language. I felt that this might well put such systems out of the reach of average engineers, even those who are good in run-of-the-mill formal reasoning. It was very important to us to craft a system that had a realistic chance of being used in practice by engineers and scientists who could simply read and write mathematical proofs.

More importantly, proofs in those systems seemed to carry an inordinate amount of

[1]Throughout this document, "he" should be read as an abbreviation for "he or she".

type information that made them large and difficult to read and write. The size factor has a particularly adverse impact in applications such as PCC [52], where compactness is essential. One might object that perhaps these are simple presentation issues that could be ameliorated by simple notational changes and desugarings, but deep issues such as the undecidability of type checking in the presence of dependent types make this far from obvious. Necula and Lee [53], for example, spend about 70 pages on the subject of reducing the size of LF proofs. Their transformations are quite involved, and the resulting proofs, although substantially smaller than the original, are still large, and even less readable than the original (due to the use of "place-holder" terms that increase compactness but devastate readability). Thus it appears that there is no easy way to take type-based proofs and readily recast them in a compact, readable form. Accordingly, DPLs take an altogether different approach: they completely do away with the idea of reducing proof checking to type checking, opting instead to hide many of the details that are necessary for proof checking into an underlying semantic framework, using the abstraction of assumption bases. We will argue that this not only achieves clarity and succinctness, but also results in a more natural cognitive model of proof construction and checking.

On the other hand, systems such as the Boyer-Moore prover seemed to rely heavily on a number of Lisp-like tricks and heuristics. That might well be appropriate for their use as proof assistants, but is not suitable for our purposes (see our remarks in the next section on the difference between "precise" and "formal"). More importantly, however, instead of proof presentation and checking, such systems are more focused on the (semi-)automatic verification of conjectures, a task on which some of them were quite effective (chiefly thanks to the heuristics): the user enters the proposition P he wishes to establish, and the system sets out on an attempt to verify that P indeed holds, perhaps with some occasional guidance from the user along the way. That is also more or less the case for systems such as HOL and PVS: they are *theorem provers* more than they are *proof languages*.

What we wanted was a small but powerful formal language in which to write down proofs in a lucid and structured style; it should have an exceptionally clean and simple syntax and semantics, and it should be extensible and programmable, so as to allow for abstraction and perhaps also for proof search (theorem proving). I could not find any such system out there, so I set out to design one myself. I started out by persistently thinking, for months, about what it is that makes the natural-deduction proofs we see in math textbooks and articles "natural"; and about how humans actually check such proofs in practice. In a nutshell, the answers that eventually emerged were: assumption and eigenvariable scope, and proof composition.

Once those ideas were properly formalized (and that took many iterations), it became fairly clear how to fit them together in a uniform, simple semantic framework.

I put it all to the test by building an implementation. The resulting system was called Athena. It was the first implemented denotational proof language, for classical first-order logic; it included a higher-order strict functional programming language in the tradition of Scheme. Before long there were niceties such as polymorphic multi-sorted universes, recursive data types, equational reasoning (term rewriting), inductive reasoning, definitional extension, etc. The resulting system was remarkably simple; its formal syntax and semantics still fit in a couple of pages. It has been used to write a provably sound Prolog interpreter, to encode parts of ZF, and for other purposes.

The abstraction leap from Athena to denotational proof languages was not difficult; it began to take place even before Athena was implemented. The main common ideas were the hiding of the "assumption base", proof composition, and the separation and integration of computation and deduction. The end result was the $\lambda\mu$-calculus (not to be confused with Parigot's calculus of the same name [55], or with any previous uses of "μ" as a minimalization operator). Denotational proof languages can be seen as "syntax sugar" for the $\lambda\mu$-calculus, much in the same way that most functional programming languages can be seen (at their core, at least) as syntax sugar for the λ-calculus.

It is noteworthy that all of this research was initially regarded as a tangent to be truncated as soon as possible in order to resume work on the programming project. Of course the issues turned out to be interesting and foundational and engaging, and the challenges far from trivial, so the tangent eventually turned into my dissertation. But the original plan remains in effect: the ultimate goal is to take all this research and apply it to software engineering. That is the main future direction of this work.

1.2 Introduction

1.2.1 Problem statement

The problem that we originally set out to tackle is, in part, this: to develop a formal language for *expressing proofs* in machine-checkable form. (By "proofs" we mean proofs in classical—though perhaps multi-sorted—first-order logic, which is adequate for most purposes.) Several systems claiming to provide that were already available, so the problem was not simply to develop "a language for expressing proofs in machine-checkable form"; there were additional constraints. We required that the language must have the following features:

1. It should have *built-in support for natural deduction*. In particular, the language should provide mechanisms and linguistic abstractions that capture informal mathematical reasoning as closely as possible.

2. It should be as *readable* and *writable* as possible, so as to have a realistic chance of being *usable* by the average engineer who is familiar with mathematical reasoning (i.e., with reading and writing proofs) but may not be an expert in mathematical logic.

3. It should have a *formal syntax and semantics*. By "formal" we do not simply mean precise. The Boyer-Moore system [8], for example, comes with a fairly precise description of what will happen when this or that command is entered, but does not have a *formal* semantics in the sense of denotational [65, 66, 32] operational [58], evaluation [40, 15], or axiomatic [26, 38] semantics. An analogy from programming languages might help. By natural language standards, the original defining report of Pascal was eminently clear and careful—but did not put forth a formal semantics. After sufficient time and scrutiny it turned out that it contained several "ambiguities and insecurities" [70]. By contrast, languages such as ML have formal semantics that prescribe the meaning of all possible constructs thoroughly and unambiguously.

4. It should have a *formal theory of observational equivalence* for proofs, providing rigorous answers to questions such as: what does it mean for two proofs to be equivalent? When can one proof be substituted for another, i.e., under what conditions can one proof be "plugged in" inside another proof without changing the latter's meaning? When is one proof more efficient than another? What kinds of optimizations can be performed on proofs? When exactly is it safe to carry out such optimizations? And so on. Again we emphasize the distinction between precise and formal.

5. It should have a *fully worked out metatheory*. In particular, the language should be *proved* sound, and if possible, complete. Soundness is clearly important in order to make it impossible for the user to prove something that does not in fact hold. Completeness is also important in order to ensure that if something does hold then there exists a proof for it.

6. It should offer *abstraction mechanisms* for composing new inference rules from existing ones, breaking up large proofs into smaller pieces, formulating and using lemmas, etc.

7. It should provide for *intuitive and efficient proof checking*. Once a user has written a proof, he should be able to have the machine check it efficiently and either announce that it is sound or indicate the source of the error. Preferably, the user should understand *how* the machine checks the proof. For this to be

the case, the proof-checking algorithm should be similar to the method that a human would use to check a proof.

8. It should provide for *efficient representation of proofs*. Because formal reasoning deals with large amounts of details, formal proofs are likely to get very large. For this reason, we should desire the proofs to be as compact as possible—though not to the point where that would compromise readability and writability.

9. It should *provide for theory development and management*. It should allow the user to introduce new vocabularies, define new symbols in terms of primitives, postulate axioms, import and export symbols and theorems from other theories, and so on.

We single out two items from the above list as being of the highest priority: readability/writability and a formal semantics. The former is indispensable if the language is to have any chance of being productively used by non-experts for non-trivial projects. We think it obvious that readability and writability depend largely on whether the language is able to express informal mathematical reasoning in a fluid style. If it reads more or less like a mathematics journal, it will be readable and writable.

Finally, a formal semantics is invaluable if proofs are to be subjected to rigorous analysis. A brief look at the history of programming languages will make for a compelling analogy. Michael Gordon in his book "The denotational description of programming languages" [32] points out that "When reasoning about programs one is often tempted to use various apparently plausible rules of inference." He then goes on to point out that what looks intuitively plausible might well be wrong, and that a formal semantics is a great safeguard against specious intuitions. The same holds true for proofs—in fact more so than for programs, because proofs are even more subtle than programs. When we reason about proofs we are often tempted to draw conclusions that seem plausible but are in fact wrong. A formal semantics is an absolute necessity for thinking about proofs rigorously (and correctly!). We close this section by completing the above quote by Gordon:

> When reasoning about programs one is often tempted to use various apparently plausible rules of inference. For example, suppose at some point in a computation some sentence, $S[\mathbf{E}]$ involving one expression $\mathbf{E}$, is true; then after doing the assignment $\mathbf{x} ::= \mathbf{E}$ one would expect $S[\mathbf{x}]$ to hold (since $\mathbf{x}$ now has the value of $\mathbf{E}$). Thus if "**y** is prime" is true before doing the assignment $\mathbf{x} ::= \mathbf{y}$ then "**x** is prime" is true after it (here $\mathbf{E}$ is $\mathbf{y}$ and $S[\mathbf{E}]$ is "**E** is prime"). Although at first sight this rule seems to be intuitively correct it does not work for most real languages. [...] It has been shown

[Ligler] that for Algol 60 the assignment rule discussed above can fail to hold in six different ways.

A formal semantics provides tools for discovering and proving exactly when rules like the above one work; without a formal semantics one has to rely on intuition—experience has shown that this is not enough. For example, the designers of EUCLID hoped that by making the language clean and simple the validity of various rules of inference would be intuitively obvious. Unfortunately when they came to actually write down the rules they found their correctness was not at all obvious and in the paper describing them [London et al.] they have to point out that they are still not sure the rules are all correct. If EUCLID had been given a formal semantics then whether or not a given rule was correct would have been a matter of routine mathematical calculation.

1.2.2 Problem clarification and brief contrast with previous work

We stress that the main task we are discussing here is that of *presenting* proofs, not *discovering* them. None of the currently available systems seemed to meet all of the above criteria. Systems such as HOL [33], PVS [54], Boyer-Moore [8], LP [28], and even Isabelle [57], are theorem provers—or at least "proof assistants"—rather than proof languages. That is, their main purpose is to help the user *discover* a proof for the result that they wish to establish.[2] By contrast, as we pointed out above, our concerns begin *after* a proof has already been discovered. Our concern is with the user who already has a proof of something on paper (or in his mind), or at least a proof outline, and wishes to transcribe this into a formal, detailed notation that is lucid, has a precise meaning, preserves the structure of the informal argument, and can be checked by machine. The distinction can be illuminated with another analogy from programming: contrast program synthesis, the task of *discovering* or "extracting" an algorithm from a given specification, versus programming, the task of *expressing* an algorithm in a formal notation—i.e., in a programming language—that is lucid ("high-level"), has a precise meaning, and can be executed by machine. Proof languages of the sort we set out to develop stand to theorem provers and proof assistants in a similar relationship that programming languages stand to program synthesis tools.

There have also been attempts at proof presentation and checking proper, but they fall short on most of the criteria we listed above. Lamport [45], for instance, proposes

[2] Or in some cases simply *verify* that the result holds: the user enters a conjecture and the system computes away and eventually comes back with a positive or negative verdict ("yes, the conjecture holds", or "I'm not sure"). Resolution theorem provers are of this flavor.

a methodology for writing proofs, but his system is informal and the resulting proofs are not in machine-checkable form. There is no formal language, no theory of proof equivalence and optimization, no soundness guarantee, etc. Proofs written for the Mizar system [64], developed in Poland, are indeed machine-checkable, but do not satisfy most of our other requirements (e.g., a formal semantics or metatheory). Ontic [51] is another proof-verification system, but is based on formal ZF set theory and experience indicates that it is impractical to reduce all mathematical reasoning to the ZF axioms. Also, as far as we know, there has been no formal metatheory (soundness, completeness, etc.) for Ontic. The systems with the most similar goals to ours are those stemming from Automath [9, 10], and include Nuprl [16], the Calculus of Constructions [19, 18], and, most notably, LF [36]. These systems are largely based on the λ-calculus augmented with dependent types and kinds. They meet some of our criteria but do not do as well on others, most notably readability and writability, and efficient proof representation and checking. A more thorough discussion of LF, in particular, can be found in Chapter 2.

1.2.3 Concrete solution and subsequent abstraction

The proof language that we proceeded to develop was called Athena. It was just what we said we were looking for: a formal language for expressing classical natural-deduction-style first-order proofs in machine-checkable form that enjoyed all of the required properties—especially readability and writability. We implemented the language and went on to experiment with it: we began to develop theories in it such as ZF, accumulate lemma libraries, and so on. The language was also successfully used by another research group at MIT investigating correctness proofs for compiler transformations [62]. But long before the implementation stage we had already started to realize that the main ideas behind Athena are applicable to a wide variety of logics: zero-order, higher-order, modal, intuitionist, equational, temporal, dynamic, program logics, and so on; and more interestingly, that these ideas could capture different proof styles: natural deduction as well as linear, "Frege-style" deduction.

Thus we started designing similar proof languages for different logics and experimenting with them. Eventually it became clear that these languages had sufficiently many things in common to constitute a natural family of languages, and thus to allow for a profitable abstraction leap. We called these languages "denotational proof languages" (DPLs), for reasons that will become clear in due course, and proceeded to study their properties in an abstract setting. We could have chosen to present Athena instead, as as a specific solution to the problem we described earlier. We felt that doing so would miss the big picture, so we decided to present the general theory of DPLs here and relegate the description of Athena to a technical report and a user manual.

As a sidelight, we should mention here an unexpected discovery that was made along the way: DPLs allow not only for proof presentation, but also for proof discovery, i.e., for theorem proving. In particular, methods, which are abstractions of proofs, allow for very powerful *proof search* mechanisms when combined with recursion and conditional branching. This means that DPLs can also be used as theorem provers, like HOL or PVS or Isabelle. We will discuss proof search in the DPL setting when we come to study the $\lambda\mu$-calculus, and we will present several examples of simple theorem provers for different logics. But by and large we will downplay this theme in order to stay focused on the subject of proof presentation and checking, where we believe our main contribution lies. While we think that the prospect of automatic proof search in DPLs is very promising and raises many interesting issues, we will only scratch the surface in this dissertation; the subject deserves to be pursued and treated on its own in a seperate document.

1.2.4 Thesis statement and discussion

Thesis statement

Our main thesis is that logics should be defined and implemented as DPLs; and that formal proofs should be written in such DPLs. We maintain that this should be the case not only for the practical task of implementing a logic on the computer so that proofs can be mechanically entered and checked, but also for the more theoretical purpose of formulating, presenting, and studying the logic, as might be done in journal articles, monographs, textbooks, lectures, private studies, etc. As justification we will shortly cite several advantages of DPLs. But before we do so we will first try to address the question that is most likely to come up at this point: What *is* a DPL? In other words, what makes a DPL a DPL?

General characteristics of DPLs

The technical answer is: a DPL is a $\lambda\mu$ system, or, more liberally, a language that can be desugared into a $\lambda\mu$ system. The precise definition of a $\lambda\mu$ system will be given in Chapter 8, and several examples will appear in Chapter 9. For now the best we can do is mention some general common traits of DPLs. Syntactically, the main characteristic is that proofs are recursively generated by context-free grammars in such a way that the text of the proof reflects what the proof actually does. Semantically, the main characteristic is that every proof has a precise meaning (or denotation; hence the name "DPL") that is formally prescribed by the semantics of the language. More importantly, the semantics are invariably based on the abstraction of *assumption bases*. In particular, *the meaning of a proof is always specified relative to a given assumption* *base*, which is a set of premises, i.e., a set of assertions that are taken as given. Intuitively, the meaning (denotation) of a proof D is the *conclusion* that D purports to establish, provided that D is not flawed. If D is erroneous then its meaning is *error*. To obtain that meaning, we *evaluate* the proof in accordance with the formal semantics of the language. The evaluation will either produce the conclusion of the proof, thus verifying its soundness, or else it will generate an error, indicating that the proof is not sound. Thus we have the DPL slogan

$$\textit{Evaluation} = \textit{Proof Checking}.$$

Note the parallel again with programming languages. There, the abstract syntax tree of an expression E such as `(* 2 (+ 3 5))` represents a computation. The formal semantics of the language assign a unique meaning to every E, representing the result of the computation (say, the number 16 for the foregoing expression). To obtain that result, we carry out the computation by evaluating E. In a DPL, the abstract syntax tree of a proof D represents a logical derivation. The formal semantics of the DPL assign a unique meaning to every D, representing the conclusion of the derivation. To obtain that conclusion, we evaluate D.

Advantages of DPLs

Below we list what we consider to be the main advantages of DPLs. For most of these we cite parts of the document that provide relevant support.

- *They are readable and writable*: DPLs avoid much of the notational and conceptual machinery that comes with general-purpose frameworks such as LF or Isabelle, e.g., higher-order abstract syntax, dependent types and kinds, higher-order unification, etc. DPL deductions are concise, easy to understand, and easy to write. The reader is invited to look at some of the many deductions we list in Chapter 6; we think he will find the notation perspicuous and succinct. Then he is urged to try to express the same deductions in the formal system of his choice and compare. We present a couple of such comparisons ourselves (in particular, see Chapter 2 for a comparison with LF).

- *They afford a high level of abstraction*: Owing to the semantics of assumption bases, DPLs make it possible to transcribe proofs at a very high level of abstraction. In addition, DPLs can provide *methods*, which can be used to abstract away from concrete deductions over any number of parameters (possibly even other methods) just as easily as functions can abstract concrete computations in higher-order functional programming languages. The reader should examine some of the methods listed in Section 9.4 that abstract away from the concrete deductions shown in Section 4.4.

- *They are easy to reason about*: See the sections on the theory of $\mathcal{CND}$, for instance (Chapter 4 and Chapter 6).
- *They are easy to optimize*. See Chapter 5 for an array of aggressive proof optimizations that go beyond the customary notion of normalization found in proof theory.
- *They are easy to learn*: For instance, any mathematician (or anyone familiar with classical first-order reasoning) should be able to pick up and start using $\mathcal{CND}$ in a day or two.
- *They are easy to implement*: Using automated lexical analysis and parsing tools such as Lex and Yacc and standard interpreter techniques, a typical DPL can be implemented in a few days.
- *They provide intuitive and efficient proof checking*: To understand how a proof is checked in a system such as LF or Isabelle one needs to understand general-purpose constraint-solving algorithms for checking dependently-typed terms. By contrast, in a DPL the proof-checking algorithm is precisely what the users themselves have been using all along—implicit manipulation of the assumption base.
- *They are programmable*: Methods can not only capture forward proof patterns; they can also express backwards proof search in a very natural style. Different "tactics" can be coded easily, without the need to maintain a stack of goals or pass around validation functions, and without the need for a type system. Soundness is automatically guaranteed by virtue of the semantics of assumption bases. See, for instance, the theorem prover we present in Section 9.5.
- *They are compact*: Compare, for instance, the sample DPL proofs in Chapter 2 with their LF equivalents.

Some of these advantages, such as ease of implementation and programmability, are especially important for the task of implementing a logic on a computer and for mechanically checking and/or discovering proofs; others, such as the facilitation of methatheory and proof theory, pertain to the more abstract task of setting up, presenting, and studying a logic; and others, such as readabily and writability, apply to both.

We will provide support for our claims by presenting and studying in detail several sample DPLs. Although we will also discuss DPLs for non-classical logics, our main focus will be classical first-order logic. All of the important issues surface in the first-order case; the extensions that are necessary for, say, a higher-order DPL, are straightforward.

Secondary thesis statement

A subsidiary claim we would like to make is that one particular DPL, $\mathcal{CND}$, which models natural first-order deduction, is, to our knowledge, the best *formal analysis* of classical mathematical reasoning to date. In order to support this claim we will discuss the nature of formal analysis in general, and then show that most popular formalizations of deductive reasoning so far have several drawbacks which our analysis avoids. The reader who is not interested in foundational issues may skip this part of the thesis.

1.2.5 Road map

We start out in Chapter 2 with a preview: a comparative example with LF.

In Chapter 3 we present the main ideas behind our approach by showing, incrementally, how to arrive at a formal DPL syntax and semantics for simple propositional logic.

In Chapter 4 we take the basic intuitions introduced in the previous chapter and develop them formally: we introduce and study $\mathcal{CND}$, a denotational proof language for "classical natural deduction" (for propositional logic). We put forth a formal theory of equivalence for $\mathcal{CND}$ deductions, and study its proof theory. We develop its metatheory, proving that it is sound and complete, and we consider some alternative semantics for it. We end the chapter by motivating the need for a "functional deduction language", paving the way for the $\lambda\mu$-calculus.

In Chapter 5 we formulate and study optimization procedures for $\mathcal{CND}$.

In Chapter 6 we extend $\mathcal{CND}$ to predicate logic with equality. We present a variety of sample deductions and argue that they faithfully capture the structure of informal mathematical arguments. We develop the theory of first-order $\mathcal{CND}$ and prove that the language is sound and complete.

In Chapter 7 we consider $\mathcal{CND}$ as a formal analysis of deduction and compare it to previous formalizations.

In Chapter 8 we show that DPLs can be understood as a class of languages based on a uniform theoretical framework, the *$\lambda\mu$-calculus*, a formal system that integrates computation and deduction. Specifically, much in the same way that different functional languages can be seen as "syntax sugar" for the λ-calculus, we will show that various DPLs ("$\lambda\mu$ systems"), including $\mathcal{CND}$, are essentially syntax sugar for the $\lambda\mu$-calculus.

We conclude in Chapter 9 with several examples of $\lambda\mu$ systems.

1.3 Notation

The following list outlines some notational conventions that will be used throughout this document:

Numbers We write N for the set of natural numbers, and N_+ for the set of non-zero natural numbers. For any $S \subseteq N$, the expression $\min(S)$ denotes the smallest number in S; and for finite $S \subseteq N$, $\max(S)$ denotes the largest number in S.

Conditionals We write $E \to E_1, E_2$ to mean "If E then E_1, else E_2". Thus if E is an expression that can take one of two possible values, one of which is regarded as "true" and the other as "false", then the expression $E \to E_1, E_2$ denotes the value of E_1 if the value of E is true, and the value of E_2 otherwise.

Functions The value of a function $f : A \to B$ for a certain element $a \in A$ is denoted by $f(a)$. Occasionally we use λ-notation for functions, e.g., $\lambda\, n \in N \,.\, n + n$ for the doubling function on the natural numbers.[3] The operation of function composition is denoted by $\cdot$, so that for any $f_1 : A \to B$ and $f_2 : B \to C$, the function $f_2 \cdot f_1 : A \to C$ is defined as

$$f_2 \cdot f_1 = \lambda\, x \in A \,.\, f_2(f_1(x)).$$

Moreover, for any $f : A \to B$ and elements $a \in A$, $b \in B$, we write $f[a \mapsto b]$ for the function $\lambda\, x : A \,.\, x = a \to b, f(x)$. We might also write

$$f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$$

as an abbreviation for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. For $A' \subseteq A$, the expression $f{\restriction}A'$ denotes the restriction of f to A'.

Pairs and lists We write $\langle x, y \rangle$ for the ordered pair with x and y as the left and right elements, respectively. For any $p = \langle x, y \rangle$ we set $l(p) = x$ and $r(p) = y$, so that $p = \langle l(p), r(p) \rangle$. By a list we will mean a finite sequence of objects, i.e., a function from an initial segment of N_+ to some finite collection of objects. For any $n \in N_+$, an expression of the form $[a_1, \ldots, a_n]$ represents the list

$$l : \{1, \ldots, n\} \to \{a_1, \ldots, a_n\}$$

such that $l(i) = a_i$ for $i = 1, \ldots, n$. The empty list is denoted by $[]$. We write $a{::}l$ for the list obtained by adding (concatenating) the element a in front of the list

[3] This could in principle cause some confusion since we are also using λ as part of several object languages, but in practice the context will always make our intentions perfectly clear.

l; and $l_1 \oplus l_2$ for the result of appending (joining) l_1 and l_2. We write $l_1 \sqsubseteq l_2$ to indicate that l_1 is a prefix of l_2; and $l_1 \sqsubset l_2$ to indicate that l_1 is a proper prefix of l_2 (i.e., $l_1 \sqsubset l_2$ iff $l_1 \sqsubseteq l_2$ and $l_1 \neq l_2$).

Herbrand terms Appendix A covers the basic theory of Herbrand terms and fixes some associated terminology and notation that is used in several parts of the document.

Additional notational conventions will be introduced in the sequel as the need arises.

"Comparisons are odious."
Christopher Marlowe, Lust's Dominion

2
A comparative example

This chapter presents a sample proof in a denotational proof language, in order to give the reader a preview of the basic themes. For comparison purposes the proof is also presented in LF, exactly as it appears in a paper by Necula and Lee [53] (the example is taken from that paper). The bulk of the chapter is devoted to an overview and critical discussion of LF, for the benefit of the readers who are not familiar with it.

2.1 A toy logic for a fragment of arithmetic

Necula and Lee [53] present a toy logic for deriving elementary arithmetic facts. We will refer to this logic as $\mathcal{TL}$.

$\mathcal{TL}$ has two main syntactic categories, expressions e and propositions P, with the following abstract syntax:

$$e \ ::= \ x \mid 0 \mid s(e) \mid e_1 + e_2 \mid e_1 - e_2$$
$$P \ ::= \ \mathbf{true} \mid e_1 \approx e_2 \mid e_1 \not\approx e_2 \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid (\forall\, x)\, P$$

where x ranges over a countably infinite set of variables. Expressions are meant to denote integers, where we intuitively interpret s, $+$, and $-$ as the successor, addition, and subtraction functions, respectively. A proposition is either the constant **true**, or an equality $e_1 \approx e_2$ or inequality $e_1 \not\approx e_2$, or a conjunction, implication, or universal quantification. Free and bound occurences of variables are defined as usual. We write

$$\frac{}{\Psi \vdash \mathbf{true}}\;[\texttt{ax1}] \qquad \frac{}{\{P\} \vdash P}\;[\texttt{ax2}] \qquad \frac{\Psi \vdash P}{\Psi \cup \Psi' \vdash P}\;[\texttt{mon}]$$

$$\frac{\Psi \vdash P_1 \quad \Psi \vdash P_2}{\Psi \vdash P_1 \wedge P_2}\;[\texttt{and-i}] \qquad \frac{\Psi \vdash P_1 \wedge P_2}{\Psi \vdash P_1}\;[\texttt{and-el}] \qquad \frac{\Psi \vdash P_1 \wedge P_2}{\Psi \vdash P_2}\;[\texttt{and-er}]$$

$$\frac{\Psi \vdash P_1 \Rightarrow P_2 \quad \Psi \vdash P_1}{\Psi \vdash P_2}\;[\texttt{mp}] \qquad \frac{\Psi \cup \{P_1\} \vdash P_2}{\Psi \vdash P_1 \Rightarrow P_2}\;[\texttt{if-i}]$$

$$\frac{\Psi \vdash (\forall\, x)\, P}{\Psi \vdash P[e/x]}\;[\texttt{spec}] \qquad \frac{\Psi \vdash P}{\Psi \vdash (\forall\, x)\, P}\;[\texttt{gen}]$$
$$\text{whenever } x \notin FV(\Psi).$$

Figure 2.1: Inference rules for a fragment of predicate logic.

$FV(P)$ for the set of variables that occur free in P, and for a set of propositions Ψ, $FV(\Psi)$ for the set of variables that occur free in some member of Ψ. Propositions that differ only in the names of their bound variables are identified. We write $P[e/x]$ for the proposition obtained from P by replacing every free occurence of x by e (taking care to rename bound variables so as to avoid name clashes).

There are two sets of inference rules. One deals with standard predicate logic, featuring introduction and elimination rules for the two connectives and the quantifier; the other pertains to arithmetic, comprising rules that capture some basic properties of addition, subtraction, and equality.

Necula and Lee present the $\mathcal{TL}$ rules in tree form. Because of the severe defects of proof trees as a formal model of natural deduction, especially in connection with assumption introduction and discharge and with eigenvariable conditions of quantifer rules (see our critique of proof trees in Section 7.2.3), it would be inordinately cumbersome and pedantic to give a rigorous definition—in the lines of the "proof expression" language in Section 4.1 of "A framework for defining logics" [36], or Section 1.2, Part B of Prawitz's monograph [59]—of $\mathcal{TL}$ in a proof-tree format. Accordingly, Necula and Lee give an informal account, using ellipses in their formulation of the inference rules and altogether evading the issue of using markers to tag assumption occurences and discharges and the role of parameters in quantifier reasoning. The drawback of an informal presentation of the object logic's proof theory is that the adequacy theorem for deductions cannot be formally stated and proved. (The reader should not be troubled if he does not understand this paragraph; he should come back to it after

$$\frac{}{\Psi \vdash e \approx e}\ [\texttt{ref}] \qquad \frac{\Psi \vdash e_1 \approx e_2}{\Psi \vdash e_2 \approx e_1}\ [\texttt{sym}]$$

$$\frac{\Psi \vdash e_1 \approx e_2 \qquad \Psi \vdash e_2 \approx e_3}{\Psi \vdash e_1 \approx e_3}\ [\texttt{tran}] \qquad \frac{}{\Psi \vdash e + 0 \approx e}\ [\texttt{id}]$$

$$\frac{\Psi \vdash e_1 \approx e_2 \qquad \Psi \vdash e_1' \approx e_2'}{\Psi \vdash e_1 + e_1' \approx e_2 + e_2'}\ [\texttt{+cong}] \qquad \frac{\Psi \vdash e_1 \approx e_2 \qquad \Psi \vdash e_1' \approx e_2'}{\Psi \vdash e_1 - e_1' \approx e_2 - e_2'}\ [\texttt{-cong}]$$

$$\frac{}{\Psi \vdash e_1 + e_2 \approx e_2 + e_1}\ [\texttt{com}] \qquad \frac{}{\Psi \vdash e - e \approx 0}\ [\texttt{inv}]$$

$$\frac{}{\Psi \vdash (e_1 + e_2) + e_3 \approx e_1 + (e_2 + e_3)}\ [\texttt{+assoc}]$$

$$\frac{}{\Psi \vdash (e_1 + e_2) - e_3 \approx e_1 + (e_2 - e_3)}\ [\texttt{+-assoc}]$$

Figure 2.2: Inference rules for a fragment of arithmetic.

finishing the chapter.) In order to avoid the complications of proof trees without sacrificing precision, we chose to present $\mathcal{TL}$ in sequent style.[1] The provability relation $\vdash$ defined by our sequent presentation is extensionally identical to that defined by the tree presentation of Necula and Lee. However, the styles are different (sequents versus trees), and it should be kept in mind that the LF declarations we introduce later in this chapter are meant to encode $\mathcal{TL}$ proof *trees*. Therefore, the reader should consult the tree-based formulation of $\mathcal{TL}$ given in the cited paper; the sequent formulation we give here is only meant as an alternative, mathematically precise presentation of the logic, to complement the informal presentation of Necula and Lee.

The rules establish judgments of the form $\Psi \vdash P$, where Ψ is a set of propositions. A judgment of this form asserts that the proposition P is provable from the propositions in Ψ. The standard predicate-logic rules are shown in Figure 2.1. The arithmetic rules appear in Figure 2.2.

The proposition of interest in this chapter is:

$$(\forall\, e_1)\,(\forall\, e_2)\,(\forall\, e_3)\; e_1 \approx e_2 + e_3 \Rightarrow e_1 - e_3 \approx e_2 \tag{2.1}$$

For an immediate comparison, the reader may contrast the LF proof of 2.1, shown in

[1]Sequents have their problems too (see Section 7.2.2), so ideally we would present $\mathcal{TL}$ as a DPL right from the start, without bothering either with proof trees or with sequents. But the purpose of this chapter is to present an introductory example, so we need to lay some initial groundwork.

Figure 2.4, page 35, with an equivalent DPL proof shown in Figure 2.9, page 44. In what follows we will discuss these proofs in more detail, first for LF and then for the DPL.

2.2 The proof in LF

It is necessary to give some background on LF before we present the proof of 2.1. While a detailed presentation of the system is beyond the scope of this chapter, we will discuss the main ideas in the next section.

2.2.1 An overview of LF

LF is a type theory: on one hand we have a well-defined set of terms and on the other we have a well-defined set of types, where the former are related to the latter via *typing judgments* of the form

$$\Gamma \vdash_\Sigma M : A \tag{2.2}$$

asserting that relative to the signature Σ and the context Γ (to be defined shortly), the term M has type A. As a formal system, LF is no more and no less than a collection of rules that specify exactly when such judgments hold. (There are rules for deriving judgments of a few other forms as well, e.g., $\Gamma \vdash_\Sigma A : K$, asserting that the kind of A in Γ and Σ is K, but for our purposes the most important judgments are those of the form 2.2.)

Specifically, the abstract syntax of LF is:

$$\begin{array}{rlll}
\text{Kinds} & K & ::= & \texttt{type} \mid \Pi x : A.K \\
\text{Types} & A & ::= & a \mid a M_1 \cdots M_n \mid \Pi x : A_1.A_2 \\
\text{Terms} & M & ::= & c \mid x \mid \lambda x : A.M \mid M_1 M_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A
\end{array}$$

where a single dot $\cdot$ represents an empty context or signature. The core of the system is the layer of terms M, which constitutes a typed λ-calculus in the style of Church (where λ-bound variables are explicitly typed). By a *phrase* we will mean either a term M, or a type A, or a kind K.

In the simplest case, a signature Σ introduces a set of types a (say, `int:type`, `prop:type`), and a set of typed constants c (say, `0:int`, or `true:prop`, `add: int -> int -> int`.)[2] The formal theory of LF comprises a set of rules which prescribe when

[2]Strictly speaking there are no types of the form $A_1 \to A_2$, but (for reasons we will explain later) such a type is viewed as syntax sugar for the type $\Pi x : A_1.A_2$ where x does not appear inside A_2.

certain judgments hold, most notably, typing judgments of the form $\Gamma \vdash_\Sigma M : A$. For instance, if Σ included the foregoing constants then we would be able to show

$$x : \texttt{int} \vdash_\Sigma \texttt{(add } x \texttt{ 0)} : \texttt{int}$$

where we write $\texttt{(add } x \texttt{ 0)}$ as an abbreviation for the curried application $\texttt{((add } x\texttt{) 0)}$. We will write *Terms*$(A)$ for the set of all terms that can be shown to have type A (relative to some signature Σ and context Γ).

Now proof representation and checking enter the picture as follows: proofs are represented as terms, and proof-checking is reduced to to type-checking. This is possible because the system is rich enough to encode the syntactic categories (expressions, formulas, etc.) and deductions of almost any given object logic $\mathcal{L}$ as typed LF terms (in our running example the object logic will be $\mathcal{TL}$). Roughly, the general procedure is this: for each class C of entitites of $\mathcal{L}$ that we wish to represent (C could be the class of formulas of $\mathcal{L}$, the class of deductions of $\mathcal{L}$, etc.) we introduce an LF type A_C, the idea being that the various elements of C will be represented by LF terms of type A_C. Thus the representation scheme can be seen as a computable mapping $[\![\cdot]\!] : C \to \mathit{Terms}(A_C)$ which assigns to each object $s \in C$ an LF term $[\![s]\!]$ of type A_C. Initially of course the type A_C will be empty, so in order to be able to build terms of this of type we must introduce constructors (constant functions symbols) with range A_C. These are usually given in curried form, so the typical signature of such a constructor will be

$$f : \cdots \to \cdots \to \cdots \to A_C. \tag{2.3}$$

We might say that such constructors "populate" or "generate" the type A_C.

How do we know how many such constructors to introduce and exactly what signature should each have? There is no algorithm for answering this question (and in fact it is here where the LF user has to employ creativity and heuristics), but the main guideline is this: we look at how the objects of C are constructed in $\mathcal{L}$, and for each way of building such an object we introduce an LF constructor of the form 2.3 that mimics the relevant production method. Thus we need to examine the *structure* of C. Often C will be an inductive closure of some sort, i.e., a minimal subalgebra generated by some initial basis, so the idea is to make A_C into a similar algebra by introducing appropriately typed constructors, and for the representation mapping $[\![\cdot]\!] : C \to \mathit{Terms}(A_C)$ to be a morphism (ideally we would like an isomorphism but LF is too rich to allow this: the set *Terms*(A_C) will be overpopulated, i.e., it will properly contain the set of images of $[\![\cdot]\!]$; see the relevant remark in page 28).

Simple LF types are usually sufficient for representing the syntactic categories of the object language, although the associated constructors often need to be higher-order (receiving arguments of functional types) in order to deal with variable-binding constructs such as quantifiers (see the discussion of higher-order abstract syntax below).

But the class of the deductions of $\mathcal{L}$ must almost invariably be represented by a *type family*, which can be viewed as an LF type indexed by terms. As a general example of this idea, outside of LF, consider the type *String*, which can be intuitively understood as the set of all finite sequences of symbols drawn from some alphabet. If we partition these sequences according to their length then we may consider "types" such as *String*(5), which contains the strings of length 5, *String*(0), which contains the (only) empty string, and so on. In this light we can regard *String* as the *type family*

$$\{String(n) \mid n \in N\}.$$

That is, we view *String* as being indexed by natural numbers, or, more formally, as a function that maps a given number (say 5) to a particular type (*String*(5)). Alternatively, we might say that *String* is a "type constructor". In LF we can introduce such type constructors with declarations like `string:int -> type`. Then a phrase such as `(string 5)` denotes a particular type. This adds a new degree of complexity to the type system. In the simply typed λ-calculus, types have a very simple structure: a type τ is either a primitive or of the form $\tau_1 \to \tau_2$, for arbitrary τ_1, τ_2. But now types can be obtained from various constructors of different signatures, so we need to specify exactly which types are to count as valid in order to weed out nonsense like `(string true nil)`—much in the same way that in the simply typed λ-calculus we have to specify which terms are well-typed in order to rule out nonsense like `(append` $\lambda x\,.\,x$`)`. This the role of *kinds*. We say that the kind of `string` is `int -> type`, and we specify that if a term M has type `int` then `(string` M`)` is a valid type.[3] Thus kinds classify types in the same spirit in which types classify terms.

By viewing a conventional type T as a type family indexed by certain objects, it becomes possible to assign much more informative types to functions operating on T. For example, consider the binary function *add*, which takes a character c and a string s and produces the string obtained by inserting c in front of s. Normally we would express the type of *add* (in curried form) as $Char \to String \to String$. Under the type-family viewpoint we can say more, namely, that *add* has the type

$$Char \to String(n) \to String(n+1) \tag{2.4}$$

which is clearly a much more refined description.

Function types such as 2.4 are called *dependent*, because the exact type of the result depends on the value of n. Admitting dependent types into a type system clearly results in increased descriptive power, but usually there is a price to be paid: type-checking

[3] Note that this introduces a mutual recursion in the specification of valid terms and valid types: whether a term M is valid may depend on whether a type A is valid, and whether a type is valid may depend on whether a term is valid.

ceases to be decidable. LF and other similar systems manage to preserve decidability by requiring all the necessary information to appear explicitly as part of the type, but we argue that this comes at the expense of readability. Decidability is salvaged, but the notation becomes cumbersome—nothing ever comes for free. In particular, a type such as 2.4 is written out in more detail by explicitly adding the parameter n up front as an additional argument, bound by the *dependent function type constructor* Π:

$$\Pi n : int \,.\, Char \rightarrow String(n) \rightarrow String(n+1). \tag{2.5}$$

We stress that in the above expression n is a bound variable.

Thus now *add* is a function of *three* arguments: a number, a character, and a string. Applying *add* to, say 5, results in a function whose type is obtained from the body of 2.5 by replacing every free occurence of n by 5:

$$Char \rightarrow String(5) \rightarrow String(5+1).$$

But the expressions $5+1$ and 6 obviously have identical values, so we should be able to "reduce" the above type to

$$Char \rightarrow String(5) \rightarrow String(6).$$

This illustrates a major complication brought about by dependent types: evaluation. If two expressions E_1 and E_2 have the same value (are reducible to the same normal form, in more operational terms) then we clearly ought to regard types like $String(E_1)$ and $String(E_2)$ as identical. Unfortunately, deciding whether two expressions are reducible to a common normal form is undecidable in any Turing-complete language. This entails that if the language is Turing-complete, as most practical programming languages are, then type checking will be undecidable if we admit dependent types. However, LF is based on the simply typed λ-calculus, which is well-known to be Turing-incomplete (the universal recursion operators of the untyped λ-calculus, in particular, are not expressible because the required self-application is not typable). Therefore, like the simply-typed λ-calculus, LF is "normalizing": every valid phrase (term, type, or kind) can be reduced to a unique normal form (in fact every valid phrase can be converted to a slightly stronger "canonical form", which is important for establishing the adequacy of LF representations). This means that we can effectively decide whether two types are equal ("definitionally equal", in LF terminology) by reducing each to its canonical form and checking that the results are identical (up to α-conversion).

Typically, we classify the deductions of $\mathcal{L}$ on the basis of their conclusions, so assuming that the conclusion of a deduction D is always a formula F of some sort,[4]

[4]Or, more generally, a *judgment* of some sort.

we consider types of the form *Deduction*(F)—the type of all deductions that derive F. Thus we view the class of deductions of $\mathcal{L}$ as the type family

$$\{Deduction(F) \mid F \in \mathbf{Form}_{\mathcal{L}}\}$$

where we write $\mathbf{Form}_{\mathcal{L}}$ for the set of formulas of $\mathcal{L}$. Of course on the assumption that $\mathcal{L}$ is consistent, i.e., that not all formulas are deducible, some of these types will be empty. For instance, if $\mathcal{L}$ is classical logic then the type *Deduction*$(P \wedge \neg P)$ will be uninhabited, since there are no deductions of $P \wedge \neg P$.

Hence, assuming that we have already encoded the formulas of $\mathcal{L}$ as terms of a type `formula`, we can introduce a type family for the deductions of $\mathcal{L}$ with a declaration such as `deduction:formula -> type`. The next step is to populate this type, i.e., to introduce constructors that can build deductions, namely, terms with types of the form `(deduction` $[\![F]\!]$`)`. As we mentioned earlier, to do this we must look at how deductions are built in $\mathcal{L}$ and try to introduce LF constructors that mimic those production methods. Now in the proof-tree model a deduction D is built by applying an inference rule R to a list of deductions $D_1, \ldots, D_n$, where this list might be empty ($n = 0$) if the rule R is an axiom (those are the base cases of the construction, i.e., the leaves of the proof tree[5]), and where the conclusions of $D_1, \ldots, D_n$ are the premises of R. So for each inference rule R of $\mathcal{L}$ we need to introduce an LF constructor f_R that takes $n \geq 0$ deductions of formulas $F_1, \ldots, F_n$ as inputs, and produces a deduction of whatever conclusion is obtained by applying R to the premises $F_1, \ldots, F_n$. A typical example is given by the rule of "$\wedge$-introduction" in customary logic, which, in proof-tree terms, takes a deduction of a formula F and a deduction of a formula G and produces a deduction of $F \wedge G$. Thus, assuming that in our encoding of formulas we have introduced a constructor `and:formula -> formula -> formula` that builds conjunctions, we can encode the said rule with a constructor `and-intro` declared as

```
and-intro:  (deduction F) -> (deduction G) -> (deduction (and F G))
```
(2.6)

Then, assuming a declaration `true:formula` and that we have an axiom for introducing `true`, say

```
true-intro:  (deduction true)
```

the following term would represent a deduction of $\mathbf{true} \wedge \mathbf{true}$:

$M =$ `(and-intro true-intro true-intro)`

where the reader can verify that the type of M is `(deduction (and true true))`.

[5]Leaf nodes might also be assumptions, or in the LF world, proofs of assumptions; thus an assumption F will be represented by a free variable $x : Deduction(F)$. Note here the underlying constructive connotation whereby we do not simply assume that the premises are true but rather that we have *proofs* of them.

Unfortunately, as we discussed above, we cannot introduce `and-intro` with a simple declaration of the form 2.6;[6] we have to add the variables F and G as explicit arguments, bound by Π, just as we had to do with the variable n in the case of *add*, in 2.5. Thus we must declare:

$$\texttt{and-intro} : \Pi F : \texttt{formula} \,.\, \Pi G : \texttt{formula} \,. \\ \texttt{(deduction F) -> (deduction G) -> (deduction (and F G))} \tag{2.7}$$

in order to ensure decidable type-checking. Thus $\wedge$-introduction is now a rule of *four* arguments, which is rather unintuitive given that we usually think of this rule as binary. In fact dependently typed inference rules with an excessive number of arguments is a general characteristic of LF that adversely impacts the size and readability of deductions because of the plethora of information that must be supplied every time a rule is applied; Figure 2.4 is a case in point. This marks a fundamental difference with DPLs. In a DPL, an inference rule such as $\wedge$-introduction would be binary, and furthermore, it would operate on formulas rather than deductions: given F and G as arguments, it would produce $F \wedge G$. Soundness is ensured simply by checking that the premises F and G are in the assumption base. This is simpler, more intuitive, and results in smaller, cleaner deductions, as manifested in Figure 2.9.

So the idea is that by introducing a signature Σ containing types such as `formula`, type families such as `deduction`, and appropriate constructors populating these types, we can define a representation mapping $[\![\cdot]\!]$ from the formulas and deductions of $\mathcal{L}$ to LF terms of the corresponding types such that:

For every formula F of $\mathcal{L}$ there is a context Γ_F, effectively obtainable from F, such that

$$\Gamma_F \vdash_\Sigma [\![F]\!] : \texttt{formula}.$$

And for every deduction D of $\mathcal{L}$ that derives a formula F from a set of $n \geq 0$ assumptions $\Psi = \{F_1, \ldots, F_n\}$, there is a context $\Gamma_{D,\Psi}$, effectively obtainable from D and Ψ, such that

$$\Gamma_{D,\Psi} \vdash_\Sigma [\![D]\!] : \texttt{(deduction } [\![F]\!]\texttt{)}.$$

As we mentioned earlier, we would like converse statements to obtain as well, e.g., that for every LF term M of type `formula` there is a unique F such that $[\![F]\!] = M$, but this is complicated by the notions of definitional equality and $\beta\eta$-interconvertibility.

[6] In an *implementation* of LF, such as Elf, we might be able to introduce rules in schematic form such as 2.6 and let the system try to *reconstruct* the full type 2.7. However, this reconstruction problem is undecidable so, in general, we have to introduce the rules in the long-hand form of 2.7. In any event, the essential point remains that the rule must be represented by a constructor of four arguments, regardless of whether the user indicates this explicitly or whether the system infers it.

We will discuss these issues in more detail soon, but for now we simply observe that if we have arrived at such a signature Σ and representation mapping $[\![\cdot]\!]$, then a proof-checking problem of the form “Does D derive F from Ψ?” may be alternatively posed as “Does the typing judgment $\Gamma_{D,\Psi} \vdash_\Sigma [\![D]\!] : (\texttt{deduction } [\![F]\!])$ hold?”. This constitutes an effective problem reduction thanks to the fact that type checking in LF is decidable. Therefore, we can mechanically decide whether a judgment such as $\Gamma_{D,\Psi} \vdash_\Sigma [\![D]\!] : (\texttt{deduction } [\![F]\!])$ holds, and hence, by the above reduction, determine whether D is indeed a deduction of F from Ψ.

There is no algorithm for obtaining the desired signature and representation mapping. There is only a set of heuristics deriving from previous experience; the users must examine how certain well-known logics have been encoded in LF and adapt those techniques to their own needs. We illustrate below using the logic $\mathcal{TL}$ as a concrete example.

2.2.2 Encoding $\mathcal{TL}$ in LF

Encoding abstract syntax

$\mathcal{TL}$ has two syntactic categories, expressions and propositions, so we introduce corresponding types

```
exp:type
prop:type
```

for each. First we will encode expressions. Recall that the abstract syntax of expressions is

$$e \ ::= \ x \mid 0 \mid s(e) \mid e_1 + e_2 \mid e_1 - e_2$$

so there are five cases to consider. It is a general and essential characteristic of LF that the variables of the object language are directly represented by LF variables, so we do not have to do anything special to encode expressions of the form x. For the remaining four cases we introduce the following constructors:

```
0 : exp
s : exp -> exp
+ : exp -> exp -> exp
- : exp -> exp -> exp
```

We can now define the representation mapping $[\![\cdot]\!]$ from $\mathcal{TL}$ expressions to LF terms of type `exp` as:

$$[\![x]\!] = x \tag{2.8}$$
$$[\![0]\!] = \texttt{0} \tag{2.9}$$
$$[\![s(e)]\!] = (\texttt{s}\ [\![e]\!]) \tag{2.10}$$
$$[\![e_1 + e_2]\!] = (\texttt{+}\ [\![e_1]\!]\ [\![e_2]\!]) \tag{2.11}$$
$$[\![e_1 - e_2]\!] = (\texttt{-}\ [\![e_1]\!]\ [\![e_2]\!]) \tag{2.12}$$

Thus, for example, the expression $0 + s(x)$ is represented by the (open) LF term `(+ 0 (s` x`))`. It is easy to show that relative to the context $x : \texttt{exp}$, this term has type `exp`. Note that the two occurences of x on the two sides of equation 2.8 stand for different things: a $\mathcal{TL}$ variable on the left side and an LF variable on the right.

We continue with propositions. Their abstract syntax is given by

$$P ::= \textbf{true} \mid e_1 \approx e_2 \mid e_1 \not\approx e_2 \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid (\forall\, x)\, P$$

so there are six alternatives to consider. We introduce a constructor for each possible case:

```
=     :  exp -> exp -> prop
<>    :  exp -> exp -> prop
true  :  prop
and   :  prop -> prop -> prop
impl  :  prop -> prop -> prop
forall:  (exp -> prop) -> prop
```

With this signature we can define the representation mapping $[\![\cdot]\!]$ from $\mathcal{TL}$ propositions to LF terms of type `prop` as follows:

$$
\begin{aligned}
[\![\textbf{true}]\!] &= \texttt{true} \\
[\![e_1 \approx e_2]\!] &= (\texttt{=}\ [\![e_1]\!]\ [\![e_2]\!]) \\
[\![e_1 \not\approx e_2]\!] &= (\texttt{<>}\ [\![e_1]\!]\ [\![e_2]\!]) \\
[\![P_1 \wedge P_2]\!] &= (\texttt{and}\ [\![P_1]\!]\ [\![P_2]\!]) \\
[\![P_1 \Rightarrow P_2]\!] &= (\texttt{impl}\ [\![P_1]\!]\ [\![P_2]\!]) \\
[\![(\forall\, x)\, P]\!] &= \texttt{forall}\ (\lambda\, x : \texttt{exp}\,.\, [\![P]\!])
\end{aligned}
$$

The first five cases are similar and straightforward, but the treatment of universal quantifications has a distinct flavor due to the higher-order type of the constructor `forall`. This is a typical example of a general methodology that advocates the use of the simply typed λ-calculus for encoding the abstract syntax of an arbitrary object language. This methodology, which goes back to Church and his formulation of type theory [14], is known as *higher-order abstract syntax*, and its key idea is to directly represent a variable x of the object language by a variable of the same name x in the λ-calculus. Then bound variables in the object language become bound by λs in the λ-calculus. In our case, consider the proposition $(\forall\, x)\, x \approx x$, where the variable x is bound by the quantifier $\forall$. This proposition is represented by the term

$$\texttt{forall}\ \ (\lambda\, x : \texttt{exp}\,.\,(\texttt{=}\ \ x\ \ x))$$

whose type, as the reader will verify, is `prop`, and where x is bound by a λ. In general, the idea is to represent every syntactic construct in the object language that binds variables (quantifiers, **let**s, patterns in function definitions, etc.) by a higher-order constant in the λ-calculus . For example, if our object language contained existential quantifications $(\exists\, x)\, P$ as well, then we would represent those with a higher-order constructor `exists` of type

```
exists:(exp -> prop) -> prop
```

and then a formula such as $(\exists\, y)\, y \not\approx 0$ would be represented by the term

$$\texttt{exists}\ \ (\lambda\, y : \texttt{exp}\,.\,(\texttt{<>}\ \ y\ \ \texttt{0})).$$

Thus all variable-binding mechanisms of the object language are modelled by the one and only variable-binding mechanism of the typed λ-calculus: λ-abstraction.

This scheme has two main advantages. First, α-equivalence at the object level is directly reflected at the λ-calculus level: if two syntax trees of the object language are alphabetic variants then the λ-calculus terms that represent them will also be alphabetic variants. Consider, for example, the propositions $(\forall\, x)\, x \approx x$ and $(\forall\, z)\, z \approx z$. These are represented by the terms $\texttt{forall}\ \ (\lambda\, x : \texttt{exp}\,.\,(\texttt{=}\ \ x\ \ x))$ and $\texttt{forall}\ \ (\lambda\, z : \texttt{exp}\,.\,(\texttt{=}\ \ z\ \ z))$, which are clearly α-equivalent. Secondly, syntactic substitution in the object language (the operation of replacing every free occurence of a variable by some syntactic object in a way that avoids variable capture) can be achieved by β-reduction in the λ-calculus. For example, in predicate logic the inference rule of universal specialization requires that we replace every free occurence of a variable x in the body P of a theorem of the form $(\forall\, x)\, P$ by some given expression e; care must be taken to ensure that the replacement does not accidentally bind any variables occuring in e. In the LF encoding given above the theorem $(\forall\, x)\, P$ would be represented by the term $\texttt{forall}\ \ (\lambda\, x : \texttt{exp}\,.\,[\![P]\!])$, so the said substitution can be properly carried out by applying the abstraction $\lambda\, x : \texttt{exp}\,.\, [\![P]\!]$ to the term $[\![e]\!]$. In this way many tedious syntactic manipulations at the object language level get relegated to the representation framework—the λ-calculus.

On the other hand there are certain disadvantages. First, although we can usually reason inductively about abstract syntax at the object level, we cannot do so at the λ-calculus level on account of the higher-order constructors. On a related note, manipulating higher-order syntax trees via operations such as pattern matching and unification is difficult, since higher-order unification and matching are intractable (either undecidable or computation-intense). By contrast, a first-order syntax representation lends itself to expressive patterns and efficient matching, which are invaluable in practice for writing derived inference rules ("methods", in DPL terminology) and theorem provers. For instance, in Athena it is possible to match propositions against patterns such as

```
(forall x (= x x))
```

or even

```
(forall [x y] (if (not P) (exists z Q))).
```

The first pattern would match propositions such as $(\forall\, z)\, z = z$ or $(\forall\, u)\, u = u$ (resulting in the bindings $\texttt{x} \mapsto z$ and $\texttt{x} \mapsto u$, respectively), but not, say, $(\exists\, y)\, R(y)$ or $(\forall\, x)\, x = y$; while the second pattern would match a proposition such as

$$(\forall\, u)\, (\forall\, w)\, [\neg R(u, w) \Rightarrow (\exists\, v)\, R(w, v)]$$

with the bindings $\texttt{x} \mapsto u$, $\texttt{y} \mapsto w$, $\texttt{P} \mapsto R(u, w)$, $\texttt{z} \mapsto v$, $\texttt{Q} \mapsto R(w, v)$. There are two advantages here: First, the patterns are succinct and perspicuous; the presence of λs and types would clutter them. Second, matching such patterns is very efficient; not so for higher-order patterns.

As another example consider metaprogramming, where we need to represent programs as first-class objects and then reason about and manipulate those objects. One thing we might want to do is determine whether an object represents a tail-recursive program. If our representation is first-order then we can do this fairly easily by matching the encoded program against a standard tail-recusion pattern; but if our encoding uses higher-order abstract syntax, this task will not be as easy.

The most important drawback for our purposes, however, is that higher-order abstract syntax quickly becomes unreadable. This is so not only on account of the unwieldy appearance of higher-order encodings (see our remarks below on size), but also because, in our experience, people seem to fundamentally think of a quantifier such as $\forall$ as a binary constructor that "takes" or "expects" two things, a variable x and a proposition P, resulting in $(\forall\, x)\, P$; this seems more natural than viewing it as a unary constructor that expects a λ-abstraction. Of course these are not problems if

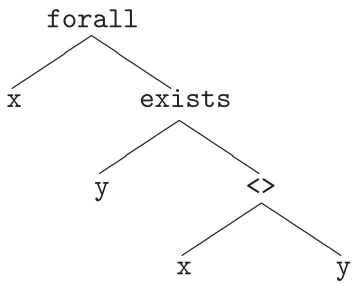


Figure 2.3: A first-order syntax tree representing the proposition $(\forall\, x)\,(\exists\, y)\, x \not\approx y$.

we intend the syntax to be manipulated by machine rather than by humans, but they are critical considerations when it comes to expressing proofs in a lucid form for the benefit of human readers (and writers).

Finally, another related problem is size: every variable-binding construct in the object language requires a constructor *and* at least one λ-abstraction of the form $\lambda\, x : A\,.\, M$. In a first-order representation both the λ and the type A are unnecessary. Consider, for example, the proposition $(\forall\, x)\,(\exists\, y)\, x \not\approx y$. A first-order representation of this proposition such as

$$\texttt{(forall x (exists y (<> x y)))} \tag{2.13}$$

carries no redundant information whatsoever. That is, every node of the syntax tree of 2.13, shown in Figure 2.3, is essential in determining the identity of the encoded proposition. When we contrast 2.13 with the higher-order representation

$$(\texttt{forall}\ \ (\underline{\lambda}\, x : \underline{\texttt{exp}}\,.\,(\texttt{exists}\ \ (\underline{\lambda}\, y : \underline{\texttt{exp}}\,.\,(\texttt{<>}\ \ x\ \ y)))))$$

we see that the underlined data are extraneous. Thus, in general, a proposition that is stored in memory as a first-order syntax tree and has n nodes representing variable-binding constructs will require at least $2n$ extra nodes when stored as a higher-order syntax tree. This not only has a negative effect on perspicuity but is also a source of inefficiency, especially for applications that require storing and manipulating large proofs.

In summary, a framework which uses first-order syntax but makes provisions for streamlining bound-variable issues would appear to get the best of both worlds.

Finally, let us consider the adequacy of this representation. As we mentioned earlier, we would like $[\![\,\cdot\,]\!]$ to provide a bijection, so that the following result would hold (where Σ is the signature given above): for every expression e of $\mathcal{TL}$ with variables $x_1, \ldots, x_n$ we have

$$x_1 : \texttt{exp}, \ldots, x_n : \texttt{exp} \vdash_\Sigma [\![e]\!] : \texttt{exp}$$

and for every LF term M such that $\Gamma \vdash_\Sigma M : \texttt{exp}$ there is a unique expression e whose variables occur in the domain of Γ[7] and such that $[\![e]\!] = M$. Similarly, we would like to have: for every proposition P with free variables $x_1, \ldots, x_n$ we have

$$x_1 : \texttt{exp}, \ldots, x_n : \texttt{exp} \vdash_\Sigma [\![P]\!] : \texttt{prop}$$

and for every LF term M such that $\Gamma \vdash_\Sigma M : \texttt{prop}$ there is a unique proposition P whose free variables occur in the domain of Γ and such that $[\![P]\!] = M$.

Unfortunately, the converses of both of these assertions fail. For expressions, consider the closed term $M = ((\lambda\, x : \texttt{exp} \,.\, x)\ \texttt{0})$. While we obviously have $\emptyset \vdash_\Sigma M : \texttt{exp}$, a cursory inspection of the defining equations 2.8—2.12 will show that M is not the image of any expression, i.e., there is no e such that $[\![e]\!] = M$. It is here where definitional equality and canonical forms enter the picture. Although M is not the image of any expression, there is a term M' that is definitionally equal to M, i.e., inter-convertible using $\beta\eta$ reduction, written $M \equiv M'$, that *is* the image of some (unique) e. This term of course is $M' = \texttt{0}$, which is the image of 0. M' is the canonical form of M, obtained from the latter via a single β -reduction. For propositions, consider the term $M = \texttt{forall (= 0)}$. The term $\texttt{(= 0)}$ has type $\texttt{exp} \to \texttt{prop}$, thus we have $\emptyset \vdash_\Sigma M : \texttt{prop}$; yet it is readily verified that M is not the image of any proposition P under the mapping $[\![\cdot]\!]$. However, the canonical form of M, namely $M' = \texttt{forall}\ \lambda\, x : \texttt{exp} \,.\, \texttt{(= 0}\ x\texttt{)}$, obtainable from M through η-conversion, *is* the image of the (unique) proposition $P = (\forall\, x)\, 0 \approx x$. (Note that we require the canonical form of a term of type $\texttt{exp} \to \texttt{prop}$ to be of the form $\lambda\, x : \texttt{exp} \,.\, N$, where N is again canonical).

Once again the computability of canonical forms by virtue of strong normalization plays a critical role.[8] In general, we can show:

$x_1 : \texttt{exp}, \ldots, x_n : \texttt{exp} \vdash_\Sigma [\![e]\!] : \texttt{exp}$ for all e with variables $x_1, \ldots, x_n$; and whenenever $\Gamma \vdash_\Sigma M : \texttt{exp}$ and M *is in canonical form*, there is a unique e whose variables appear in Γ with type $\texttt{exp}$ and such that $[\![e]\!] = M$.

Likewise,

$x_1 : \texttt{exp}, \ldots, x_n : \texttt{exp} \vdash_\Sigma [\![P]\!] : \texttt{prop}$ for all P with free variables $x_1, \ldots, x_n$; and whenenever $\Gamma \vdash_\Sigma M : \texttt{prop}$ and M *is in canonical form*, there is a unique P whose free variables appear in Γ with type $\texttt{exp}$ and such that $[\![P]\!] = M$.

[7] By the domain of a context $\Gamma = y_1 : A_1, \ldots, y_k : A_k$ we mean the set $\{y_1, \ldots, y_k\}$.

[8] In the original formulation of LF the notion of canonical forms was based on β-reduction only. The incorporation of η-reduction, which greatly facilitates the studying of LF's representational adequacy, was made possible by later work.

An alternative way of stating the converses of such results is the following:

> If $\Gamma \vdash_\Sigma M$: `exp` then there is an expression e whose variables appear in Γ with type `exp` and such that $[\![e]\!]$ and M are definitionally equal ($\beta\eta$-interconvertible).

and likewise for propositions.

Encoding proofs

We introduce a type family of proofs, indexed by propositions:

```
pf: prop -> type
```

Next we need to introduce constructors representing the inference rules of $\mathcal{TL}$. The following declarations model the rules of Figure 2.1:

```
true_i : pf true
and_i  : ΠP:prop.ΠQ:prop.pf P -> pf Q -> pf (and P Q)
and_el : ΠP:prop.ΠQ:prop.pf (and P Q) -> pf P
and_er : ΠP:prop.ΠQ:prop.pf (and P Q) -> pf Q
impl_i : ΠP:prop.ΠQ:prop.(pf P -> pf Q) -> pf (impl P Q)
impl_e : ΠP:prop.ΠQ:prop.pf (impl P Q) -> pf P -> pf Q
all_i  : ΠP:exp -> prop.(Πe:exp.pf (P e)) -> pf (forall P)
all_e  : ΠP:exp -> prop.Πe:exp.pf (forall P) -> pf (P e)
```

The second set of constructors encode the arithmetic rules shown in Figure 2.2:

```
ref     : Πe:exp.pf (= e e)
sym     : Πe1:exp.Πe2:exp.pf (= e1 e2) -> pf (= e2 e1)
tran    : Πe1:exp.Πe2:exp.Πe3:exp.pf (= e1 e2) -> pf (= e2 e3) -> pf (= e1 e3)
id      : Πe:exp.pf (= (+ e 0) e)
com     : Πe1:exp.Πe2:exp.pf (= (+ e1 e2) (+ e2 e1))
inv     : Πe:exp.pf (= (- e e) 0)
+cong   : Πe1:exp.Πe2:exp.Πe3:exp.Πe4:exp.
            pf (= e1 e2) -> pf (= e3 e4) -> pf (= (+ e1 e3) (+ e2 e4))
-cong   : Πe1:exp.Πe2:exp.Πe3:exp.Πe4:exp.
            pf (= e1 e2) -> pf (= e3 e4) -> pf (= (- e1 e3) (+ e2 e4))
+assoc  : Πe1:exp.Πe2:exp.Πe3:exp.pf (= (+ (+ e1 e2) e3) (+ e1 (+ e2 e3)))
+-assoc: Πe1:exp.Πe2:exp.Πe3:exp.pf (= (- (+ e1 e2) e3) (+ e1 (- e2 e3)))
```

Most of these should be self-explanatory after the background of Section 2.2.1. The interesting cases are `impl_i`, `all_i`, and `all_e`, which we will discuss below. However, we should first mention that if a precise definition of $\mathcal{TL}$ deductions as proof trees were available then we could, at this point, use the above declarations to formally define the representation image $[\![D]\!]$ of any given proof tree D, and proceed to state and prove an adequacy theorem for the representation of deductions akin to the adequacy result for expressions and propositions in page 29. In practice this is important in order to guarantee that the (canonical) terms of type `pf` $[\![P]\!]$ coincide with the object proofs of P. For an example of how this may be done rigorously the reader is referred to the defining LF document [36].

The constructor `impl_i` embodies the following rule for introducing conditionals: if we have an effective method (function) for transforming a given proof of P into a proof of Q, then we may infer $P \Rightarrow Q$. (This is precisely Heyting's original constructive interpretation of implication.) Thus the higher-order signature of `iml_i`:

```
impl_i:  (pf P -> pf Q) -> pf (impl P Q)
```

or, after adding the schematic variables P and Q as explicit parameters:

$$\texttt{impl\_i} \;:\; \Pi P\texttt{:prop}.\Pi Q\texttt{:prop}.\texttt{(pf}\ P\ \texttt{->}\ \texttt{pf}\ Q\texttt{)}\ \texttt{->}\ \texttt{pf (impl}\ P\ Q\texttt{)}.$$

As an example, here is a proof of the implication $P \wedge Q \Rightarrow P \wedge P$ (for any P, Q):

```
impl_i (and [[P]] [[Q]])
       (and [[P]] [[P]])
       (λp:pf (and [[P]] [[Q]]).
          (and_i [[P]]
                 [[P]]
                 (and_el [[P]] [[Q]] p)
                 (and_el [[P]] [[Q]] p)))
```

The reader will verify that the type of this term is

$$\texttt{(impl (and}\ [\![P]\!]\ [\![Q]\!]\texttt{) (and}\ [\![P]\!]\ [\![P]\!]\texttt{))}.$$

The LF proof should be contrasted with the following DPL proof:

assume $P \wedge Q$
  dlet $l = $!left-and $P \wedge Q$
    !both l l

The constructor `all_i` encodes the following informal rule for introducing universal generalizations $(\forall\, x)\, P$: if we have a function f_P which can map any given individual a in the universe of discourse to a proof of $P(a)$, then we may infer $(\forall\, x)\, P$. Of course the universe of discourse varies according to the interpretation at hand, so we resort to syntactic means: we require the function f_P to map any given *expression* e to a proof of $P[e/x]$. Hence the signature of `all_i`:

$$\texttt{all\_i: } \Pi e\texttt{:exp.pf (}P\ e\texttt{) -> pf (all } P\texttt{).} \tag{2.14}$$

Here we reap the LF benefit of relegating substitution at the level of the object language to β -reduction in the λ-calculus. Recalling that the body $P(x)$ of a universal generalization is represented by a λ-abstraction of the form $M = \lambda\, x : \texttt{exp} \,.\, [\![P]\!]$, we see that $[\![P[e/x]]\!]$ is simply $(M\ \ [\![e]\!])$. Hence, assuming that the variable P in 2.14 stands for the body of a universal quantification, and is thus of type `exp -> prop`, we see that `all_i` is a unary constructor whose argument is precisely the function f_P discussed above, of dependent type $\Pi\, e : \texttt{exp} \,.\, \texttt{pf (}P\ \ e\texttt{)}$ (give me any expression e and I will give you a proof of $(P\ \ e)$) and whose result is a proof of the universal quantification `(all` P`)`. Of course P is free in 2.14 so we need to bind it with Π, resulting in

$$\texttt{all\_i : } \Pi P\texttt{:exp -> prop.(}\Pi e\texttt{:exp.pf (}P\ e\texttt{)) -> pf (all } P\texttt{).}$$

As an example of using this rule, here is a proof of $(\forall\, x)\, x \approx x$:

$$\texttt{all\_i (}\lambda x\texttt{:exp. (= } x\ x\texttt{)) (}\lambda e\texttt{:exp. (ref } e\texttt{)).} \tag{2.15}$$

To see that the type of this term is indeed `pf (forall (`λx`:exp.(=` $x\ \ x$`)))`, we must consult the LF rules for typing abstractions $\lambda\, x : A \,.\, M$ and applications $M\, N$, which are variations of the usual rules of the simply typed λ-calculus in order to account for dependent types. First, the rule for abstractions is

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda\, x : A \,.\, M : \Pi\, x : A \,.\, B}\ [\texttt{abs}]$$

where A is any valid type. The dual rule for typing applications of dependently typed functions is more interesting:

$$\frac{\Gamma \vdash_\Sigma M : \Pi\, x : A \,.\, B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\, N : B[N/x]}\ [\texttt{app}]$$

Thus, if M is a function of $x : A$, and the *type* of its result is also a function of the value of x, $B(x)$, then applying M to an argument N (of the right type A) will result in a value of type $B[N/x]$. Of course if the type B does not depend on the value of x, as would be the case if x does not occur free in B, then the type $\Pi\, x : A \,.\, B$ degenerates to the non-dependent type $A \rightarrow B$, and since $B[N/x]$ is then B, the above rule becomes the usual typing rule for applications

$$\frac{\Gamma \vdash_\Sigma M : A \rightarrow B \qquad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma M\, N : B}$$

while [abs] becomes the usual typing rule for abstractions:

$$\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda\, x : A \,.\, M : A \rightarrow B}$$

That is why it is not necessary to introduce types of the form $A \rightarrow B$ as primitives; such a type can be seen as syntax sugar for $\Pi\, x : A \,.\, B$ for any x that does not occur in B (note that this desugaring is uniquely defined up to α-equivalence).

Now we must use the rules [abs] and [app] to show that the type of 2.15 is

$$\texttt{pf (forall (}\lambda x\texttt{:exp.(= } x \; x\texttt{)))}$$

where the reader will recall that 2.15 is an abbreviation for the curried application

$$\texttt{((all\_i } M_1\texttt{) } M_2\texttt{)}, \text{ where } M_1 = \texttt{(}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{))}, M_2 = \texttt{(}\lambda\, e : \texttt{exp . (ref } e\texttt{))}.$$

Using [abs] and the foregoing desugaring of $\rightarrow$, the type of M_1 is readily seen to be $\texttt{exp} \rightarrow \texttt{prop}$. Therefore, using [app], we conclude that the type of $\texttt{(all\_i } M_1\texttt{)}$ is $(\Pi\, e : \texttt{exp . pf (}M_1 \;\; e\texttt{)}) \rightarrow \texttt{pf (all } M_1\texttt{)}$, i.e.,

$$\begin{aligned} \texttt{(all\_i } M_1\texttt{) :} \quad & (\Pi\, e : \texttt{exp . pf ((}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{)) } e\texttt{)}) \rightarrow \\ & \texttt{pf (all (}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{)))}. \end{aligned} \tag{2.16}$$

Now using [abs], [app], and the given type of ref, we see that the type of M_2 is $\Pi\, e : \texttt{exp . pf (= } e \;\; e\texttt{)}$:

$$M_2 : \; \Pi\, e : \texttt{exp . pf (= } e \;\; e\texttt{)}. \tag{2.17}$$

Next we must type $\texttt{((all\_i } M_1\texttt{) } M_2\texttt{)}$ using [app], 2.16, and 2.17, but we run into a problem: as can be seen from 2.16, $\texttt{(all\_i } M_1\texttt{)}$ expects an argument of type

$$\Pi\, e : \texttt{exp . pf ((}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{)) } e\texttt{)} \tag{2.18}$$

whereas M_2 has type

$$\Pi\, e : \texttt{exp . pf (= } e \;\; e\texttt{)}. \tag{2.19}$$

Thus it appears that we have a type mismatch, and that the application cannot go through.

However, the day is saved by the notion of type inter-convertibility that we motivated earlier: the terms $\texttt{((}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{)) } e\texttt{)}$ and $\texttt{(= } e \;\; e\texttt{)}$ are convertible to the same normal form via β-reduction, hence the types

$$\texttt{pf ((}\lambda\, x : \texttt{exp . (= } x \;\; x\texttt{)) } e\texttt{)} \;\; \text{and} \;\; \texttt{pf (= } e \;\; e\texttt{)}$$

are equivalent, just as the types $String(E_1)$ and $String(E_2)$ are equivalent whenever E_1 and E_2 have the same value. Hence 2.18 and 2.19 are identical types and the application `((all_i` M_1`)` M_2`)` can be successfully typed, resulting in the desired `pf (forall (`λx`:exp.(=` x x`)))`. Contrast 2.15 with the DPL proof

pick-any x
  !ref x

Finally, the constructor `all_e` acts as an inverse to `all_i`, specializing a universal quantification to a given expression. In LF this inverse relationship is conveniently captured by the corresponding inverse relationship between λ-abstraction and application. Note that the type of the result of `all_e` is `pf (`P e`)`, obtained by applying P to e. The usual provisos for avoiding variable capture are thus automatically observed. As a last example illustrating this rule, here is a proof of the proposition $(\forall\, x)\, x \approx 0 \Rightarrow s(0) \approx 0$:

```
impl_i (forall (λx:exp.(= x 0)))
       (= (s 0) 0)
       (λp:pf (forall (λx:exp.(= x 0))).
          (all_e (forall (λx:exp.(= x 0)))
                 (s 0)
                 p))
```

Contrast this with the corresponding DPL proof:

assume $P = (\forall\, x)\, x \approx 0$ **in**
  !specialize P $s(0)$

We are now ready to present the LF proof of 2.1; it appears in Figure 2.4.

Reducing the size of LF proofs

We mention in closing that in their cited paper Necula and Lee proceed to introduce an "implicit representation" for LF proofs, in which many types and terms are replaced by an unknown "place-holder" phrase `*`. Figure 2.5 shows the proof of Figure 2.4 in implicit form. Their goal is to cut down the size of proofs and speed up their validation by eliminating redundant components from LF representations. Clearly, size is an important factor for applications such as proof-carrying code [52], where large proofs may be shipped over a network; proof-checking speed is also an important consideration.

The idea is to transform a proof M into a "special" term $\widehat{M}$ containing place-holders, and then manipulate $\widehat{M}$ instead of M. This scheme involves two transformations, which may be viewed as inverses of each other:

```
all_i (λe1 : exp.
        all (λe2 : exp.
              all (λe3 : exp.
                   impl (= e1 (+ e2 e3))
                        (= (- e1 e3) e2))))
      (λe1 : exp.
        all_i (λe2 : exp.
              all (λe3 : exp.
                   impl (= e1 (+ e2 e3))
                        (= (- e1 e3) e2)))
              (λe2 : exp.
                all_i (λe3 : exp.
                       impl (= e1 (+ e2 e3))
                          (= (- e1 e3) e2))
                       (λe3 : exp.
                         (impl_i (= e1 (+ e2 e3))
                                 (= (- e1 e3) e2)
                                    (λu : pf (= e1 (+ e2 e3)).
                                      (tran (- e1 e3)
                                           (- (+ e2 e3) e3)
                                           e2
                                           (-congr e1
                                                   (+ e2 e3)
                                                   e3
                                                   e3
                                                   u
                                                   (ref e3))
                                           (tran (- (+ e2 e3) e3)
                                                (+ e2 (- e3 e3))
                                                e2
                                                (+-assoc e2
                                                         e3
                                                         e3)
                                                (tran (+ e2 (- e3 e3))
                                                     (+ e2 0)
                                                     e2
                                                     (+congr e2
                                                             e2
                                                             (- e3 e3)
                                                             0
                                                             (ref e2)
                                                             (inv e3))
                                                     (id e2)))))))))
```

Figure 2.4: The LF proof of 2.1.

```
all_i *
      (λe₁ : *.
        all_i *
              (λe₂ : *.
                all_i *
                      (λe₃ : *.
                        (impl_i * *
                                (λu : *.
                                  (=tr * * *
                                       (-congr * * * *
                                               u
                                               (=id *))
                                       (=tr * * *
                                            (+-assoc * * *)
                                            (=tr * * *
                                                 (+congr * * * *
                                                         (=id *)
                                                         (+inv *))
                                                 (+id *))))))))))
```

Figure 2.5: Necula and Lee's implicit representation of the same proof.

(i) One is an encoding transformation, which takes a given LF term M and produces a compactified pseudo-term $\widehat{M}$ ("pseudo" because it contains special place-holder tokens, which are not, strictly speaking, part of LF). This transformation is akin to type-erasure embeddings of the terms of the simply typed λ-calculus into the terms of the untyped λ-calculus.

(ii) The other is a decoding transformation that takes a pseudo-term $\widehat{M}$ and a claimed type A and retrieves M *provided that M has type A*; otherwise the transformation fails. Thus the reconstruction of M is interwoven with its validation: we can reconstruct M from $\widehat{M}$ iff M has the advertised type A.

This of course means that we can check whether a given M has type A by first computing a pseudo-term $\widehat{M}$ for it and then attempting to reconstruct M from $\widehat{M}$ and A.

For part (i) of the above process, we note that for any given M there are several different pseudo-terms $\widehat{M}$ encoding it, some more compact than others, or more conducive to part (ii) than others. Accordingly, there are several different ways of choosing

$\widehat{M}$, each with its own advantages; Necula and Lee present three different encoding algorithms. Of course the most important requirement is that $\widehat{M}$ should be in a form that allows part (ii) to be carried out mechanically.

For our purposes the issue is rather peripheral since pseudo-terms are meant to be produced and manipulated by machine, not by humans (it is clear from Figure 2.5 that such terms are neither readable nor writable). Nevertheless, we make the following observations:

- The complexity of the encoding and decoding process is super-exponential. The reconstruction algorithm, in particular, is heavily reliant on higher-order unification. However, Necula and Lee report satisfactory size and speed improvements *over regular LF proofs*.

- Even optimally compact *pseudo* LF terms are still larger than *regular* DPL proofs. For instance, the abstract syntax tree of the pseudo-term shown in Figure 2.5 has more nodes than the abstract syntax tree of the DPL proof of Figure 2.9. This is mainly because of the λs: the heavily higher-order types of LF constructors result in a large number of explicit λ-abstractions as arguments. In addition, the excessive number of arguments required by dependently typed inference rules (see page 23) results in a large number of place holders. Thus it appears that regular DPL proofs are always at least as efficient—and usually more so—as the most optimized pseudo-LF proofs; while, more significantly, they enjoy this efficiency without sacrificing readability, which is not the case for pseudo-LF proofs.

2.3 The DPL proof

Let us first see how we might go about proving 2.1 informally. We want to show that if $e_1 \approx e_2 + e_3$ then $e_1 - e_3 \approx e_2$, for arbitrary e_1, e_2, and e_3. Accordingly, let any e_1, e_2, and e_3 be given, and suppose that the equality

$$e_1 \approx e_2 + e_3 \tag{2.20}$$

holds. If we subtract e_3 from both sides of this equality, we get

$$e_1 - e_3 \approx (e_2 + e_3) - e_3. \tag{2.21}$$

(Actually this cannot be done in one step using the rules of $\mathcal{TL}$, but it can be done in two: from [`ref`] we get $e_3 \approx e_3$, and then by applying [`-cong`] to the assumption 2.20 and $e_3 \approx e_3$ we get 2.21.)

Now from the associativity rule [`+-assoc`] we have

$$(e_2 + e_3) - e_3 \approx e_2 + (e_3 - e_3) \tag{2.22}$$

so applying the transitivity rule [`tran`] to 2.21 and 2.22 gives

$$e_1 - e_3 \approx e_2 + (e_3 - e_3). \tag{2.23}$$

Thus *if we could only prove the equality*

$$e_2 + (e_3 - e_3) \approx e_2 \tag{2.24}$$

then one final application of transitivity to 2.23 and 2.24 would yield the desired $e_1 - e_3 \approx e_2$.

This is top-down problem decomposition similar to the "successive refinement" technique of structured programming: when we need to write a procedure f to solve a given problem, express f in terms of a small number of auxiliary procedures $f_1, \ldots, f_n$ that have the appropriate input/output behavior but may have not been implemented yet. Thus at this stage of the problem-solving process we are viewing each f_i as a "black box": we specify—or rather postulate—exactly *what* f_i is supposed to do, but we say nothing about *how* that might be done. This state of affairs can be depicted as a tree with f at the root and $f_1, \ldots, f_n$ as the children. Then on the next iteration we expand this tree by repeating the same process with each child f_i, until eventually all the required black boxes at the leaves are primitives offered by the language and thus do not need to be implemented.

Likewise, we have now reduced the original problem to the following task: given any e_2 and e_3, prove the equality $e_2 + (e_3 - e_3) \approx e_2$. This is easily done as follows: From reflexivity, we get

$$e_2 \approx e_2 \tag{2.25}$$

while [`inv`] gives

$$e_3 - e_3 \approx 0 \tag{2.26}$$

and so applying [`+cong`] to 2.25 and 2.26 results in

$$e_2 + (e_3 - e_3) \approx e_2 + 0. \tag{2.27}$$

But, from [`id`], we have

$$e_2 + 0 \approx e_2 \tag{2.28}$$

and therefore transitivity on 2.27 and 2.28 produces the goal $e_2 + (e_3 - e_3) \approx e_2$. Let us refer to the equality $e_2 + (e_3 - e_3) \approx e_2$ as *lemma M*.

The structure of this informal proof can be more succinctly depicted as shown in Figure 2.6. Now contrast Figure 2.6 with the DPL proof that appears in Figure 2.7. We believe that the DPL proof preserves the structure of the informal argument to such an extent that anyone who can make sense of the informal proof can also follow

<u>*Prove*</u>: $(\forall e_1)(\forall e_2)(\forall e_3)\, e_1 \approx e_2 + e_3 \Rightarrow e_1 - e_3 \approx e_2$

pick any e_1, e_2, e_3

assume (1) $e_1 \approx e_2 + e_3$

Then:	(2)	$e_3 \approx e_3$	from [`ref`]
	(3)	$e_1 - e_3 \approx (e_2 + e_3) - e_3$	from (1), (2), and [`-cong`]
	(4)	$(e_2 + e_3) - e_3 \approx e_2 + (e_3 - e_3)$	from [`+-assoc`]
	(5)	$e_1 - e_3 \approx e_2 + (e_3 - e_3)$	from (3), (4), and [`tran`]
	(6)	$e_2 + (e_3 - e_3) \approx e_2$	from lemma M
	(7)	$e_1 - e_3 \approx e_2$	from (5), (6), and [`tran`]

where lemma M proves $e_2 + (e_3 - e_3) \approx e_2$ for any given e_2 and e_3 as follows:

<u>*Prove*</u>: $e_2 + (e_3 - e_3) \approx e_2$

(1)	$e_2 \approx e_2$	from [`ref`]
(2)	$e_3 - e_3 \approx 0$	from [`inv`]
(3)	$e_2 + (e_3 - e_3) \approx e_2 + 0$	from 1, 2, and [`+cong`]
(4)	$e_2 + 0 \approx e_2$	from [`id`]
(5)	$e_2 + (e_3 - e_3) \approx e_2$	from 3, 4, [`tran`]

Figure 2.6: Structure of the informal proof of 2.1.

the DPL proof, even if they do not know anything about DPLs. We believe that the same cannot be said for the LF proof; someone who understands the informal proof shown in Figure 2.6 but does not know anything about LF will not be able to make sense of the LF proof in Figure 2.4. (In fact it might be argued that even people who are familiar with LF would not be readily able to follow Figure 2.4.)

We will not explain the DPL proof here in detail. Every aspect of it will be thoroughly discussed in the sequel. We want to draw attention, however, to the similarity between DPL proofs and functional programs. To a large extent the DPL proofs shown here read like Scheme or ML programs. This is a general characteristic of DPLs, and it is not limited to appearances; it extends to semantics. DPL proofs can be *evaluated* much in the same spirit that functional programs are evaluated. This kinship is one of the reasons why we expect DPLs to appeal to programmers; learning and using a DPL can draw on intuitions that almost all programmers have, even those who use untyped languages.

In fact the DPL methodology for developing proofs is very akin to the top-down decomposition methodology for programming that we dicsussed above. When a DPL user faces the task of proving a complex proposition P, he sets out to express the proof in terms of inference rules ("methods") $M_1, \ldots, M_n$ which have the desired premise/conclusion behavior (such that their given composition yields the goal P) but may not be primitive rules of the underlying logic. Then he proceeds to implement each M_i by expressing it in terms of other "black-box" methods, and so on until all methods at the leaves of the decomposition tree are primitive inference rules.

The auxiliary methods at the internal nodes of this tree are what many authors call "derived inference rules", because they are ultimately expressible in terms of primitive rules. The abstraction power that they afford is an indispensable tool for managing conceptual complexity; it allows us to break up large proofs into simpler modular components, resulting in smaller, cleaner, and reusable units of reasoning. We quote from Manna [47]:

> In order to be able to write shorter deductions for wwfs in practice, it is most convenient to have a library of *derived inference rules*. Each such rule can be given an effective proof in the sense that we can show effectively how to replace any derived rule of inference whenever it is used in a deduction by an appropriate sequence of wffs using only the "primitive" rules of inference and axioms.

A prime example of a derived inference rule is one that derives our "lemma M", i.e., the equality $e_2+(e_3-e_3) \approx e_2$, for any given e_2 and e_3. Although $\mathcal{TL}$ does not offer any one single rule for establishing such an equality directly, we know from Figure 2.6 that the equality can be derived by an appropriate composition of $\mathcal{TL}$ primitives. In DPL style this derivation may be expressed as the following deduction D, where e_2 and e_3 are any two *particular* expressions:

$$D = \begin{array}{l} \textbf{dlet}\ \ P_1 = e_2 \approx e_2\ \ \textbf{by}\ \texttt{!ref}\ e_2 \\ \qquad\ \ P_2 = e_3 - e_3 \approx 0\ \ \textbf{by}\ \texttt{!inv}\ e_3 \\ \qquad\ \ P_3 = e_2 + (e_3 - e_3) \approx e_2 + 0\ \ \textbf{by}\ \texttt{!+cong}\ P_1\ P_2 \\ \qquad\ \ P_4 = e_2 + 0 \approx e_2\ \ \textbf{by}\ \texttt{!id}\ e_2 \\ \textbf{in} \\ \quad e_2 + (e_3 - e_3) \approx e_2\ \ \textbf{by}\ \texttt{!tran}\ P_3\ P_4 \end{array} \tag{2.29}$$

It is clear that D in no way depends on the specific values of e_2 and e_3, and in that sense it represents a proof *schema*: by instantiating e_2 and e_3 with various specific expressions we obtain various specific proofs. There are, of course, infinitely many choices for e_2 and e_3. In the DPL world we can immediately turn such a schema into a reusable module by parameterizing over e_2 and e_3 via the *method abstraction operator* μ, arriving at a method $M = \mu\, e_2, e_3 \,.\, D$ that can be *applied* to *any* given expressions

e_2 and e_3 to derive the equality $e_2 + (e_3 - e_3) \approx e_2$. For instance, the application of M to 0 and $s(0)$ would look like $(!M\ 0\ s(0))$ and would result in $0 + (s(0) - s(0)) \approx 0$. Thus for all practical purposes we can now treat M as an indivisible, atomic inference rule. Every time we use it, of course, it will actually be "expanded out" into a series of applications of primitive $\mathcal{TL}$ rules, but the expansion will occur automatically by the DPL implementation; all the user has to do is write $(!M\ 0\ s(0))$. This is similar to λ-abstraction in programming. When we define a cubing function

$$C = \lambda\, x\,.\,(*\ x\ (*\ x\ x))$$

we can subsequently use C as if it were a primitive of the language. Of course every time we apply C to an argument, the application will be expanded out into two successive multiplications, but this is an implementation issue that can—and should—be ignored by the clients of C, who can more profitably view C as a black box that is completely determined by its input-output behavior.

Derived inference rules such as the above that are obtained from concrete proofs through schematic abstraction can also be formulated in LF-like systems through λ-abstraction. However, there is an essential difference: DPL methods have dynamic evaluation semantics (they are "executable"), whereas in the LF world proofs are static—there is no notion, for example, of an inference rule recursively calling itself. (In fact, as we mentioned earlier, the absence of unbounded recursion is of paramount importance for LF and similar systems. It must be stressed that the λ-calculus in LF is used purely for representational purposes—for encoding formulas, proofs, and so on; *not* for writing loops, conditionals, etc.) Owing to the power afforded by their ability to iterate, branch conditionally, etc., DPL methods go far beyond schematic abstraction.

For example, it is straightforward in a DPL such as Athena to write a method that replaces every subformula B_{old} of a given formula A with a formula B_{new} that is provably equivalent to B_{old}. This can be written as a method that takes A, B_{old}, and B_{new} as arguments, and provided that the equivalence of B_{old} and B_{new} holds (the biconditional $B_{old} \Leftrightarrow B_{new}$ is in the assumption base), goes on to derive the formula obtained from A by replacing every occurence of B_{old} by B_{new}. The method is recursive and proceeds by induction on the structure of A; it takes a few lines and is remarkably perspicuous. In fact in Section 9.4 we present this particular method in full detail, along with several other concrete examples. Expressing the same derived inference rule in a system such as LF would be much more complicated, requiring declarations with not just dependent types, but also with dependent kinds.

The DPL deduction in Figure 2.7 is expressed in *conclusion-annotated form*, whereby the conclusion of each method application is explicitly attached to the application with

```
pick-any e_1, e_2, e_3
  assume P_1 = e_1 ≈ e_2 + e_3
    dlet P_2 = e_3 ≈ e_3  by !ref e_3
         P_3 = e_1 − e_3 ≈ (e_2 + e_3) − e_3  by !-cong P_1 P_2
         P_4 = (e_2 + e_3) − e_3 ≈ e_2 + (e_3 − e_3)  by !+-assoc e_2 e_3 e_3
         P_5 = e_1 − e_3 ≈ e_2 + (e_3 − e_3)  by !tran P_3 P_4
         P_6 = e_2 + (e_3 − e_3) ≈ e_2  by !M e_2 e_3
    in
      e_1 − e_3 ≈ e_2  by !tran P_5 P_6
where
M = μ e_2, e_3 .
  dlet P_1 = e_2 ≈ e_2  by !ref e_2
       P_2 = e_3 − e_3 ≈ 0  by !inv e_3
       P_3 = e_2 + (e_3 − e_3) ≈ e_2 + 0  by !+cong P_1 P_2
       P_4 = e_2 + 0 ≈ e_2  by !id e_2
  in
    e_2 + (e_3 − e_3) ≈ e_2  by !tran P_3 P_4
```

Figure 2.7: A DPL proof of 2.1 in conclusion-annotated form.

the keyword **by**. For instance, instead of

$$\textbf{modus-ponens}\ P \Rightarrow Q, P \tag{2.30}$$

we write

$$Q\ \textbf{by}\ \textbf{modus-ponens}\ P \Rightarrow Q, P. \tag{2.31}$$

When the application is valid (i.e., successfully produces the conclusion Q), then 2.30 and 2.31 are perfectly equivalent: both of them yield the same result—Q. The difference is stylistic: 2.31 is more verbose, but provides more documentation and is arguably more readable than 2.30. Whether one writes in conclusion-annotated form or in the style of 2.30, which we may call *direct*, depends on one's goals (is it deemed desirable to have as much documentation as possible?) and personal taste. DPLs can accomodate both. The DPL proof of Figure 2.7 expressed in direct style appears in Figure 2.8.

Next, by inlining the method M and by expanding the **dlet**s of Figure 2.8 into nested method applications, much in the same fashion in which one might expand a **let** in ML or in Scheme into nested function applications, we arrive at the DPL proof shown in Figure 2.9. The process is quite similar to casting the Scheme code

```
(define (double l)
```

pick-any e_1, e_2, e_3
  assume $P_1 = e_1 \approx e_2 + e_3$
    dlet $P_2 =$ `!ref` e_3
        $P_3 =$ `!-cong` P_1 P_2
        $P_4 =$ `!+-assoc` e_2 e_3 e_3
        $P_5 =$ `!tran` P_3 P_4
        $P_6 =$ `!`M e_2 e_3
    in
      `!tran` P_5 P_6
where
$M = \mu\, x, y$.
  dlet $P_1 =$ `!ref` x
      $P_2 =$ `!inv` y
      $P_3 =$ `!+cong` P_1 P_2
      $P_4 =$ `!id` x
  in
    `!tran` P_3 P_4

Figure 2.8: A DPL proof of 2.1 in direct style.

```
  (append l l))

(let ((a (* 2 pi))
      (rev-lst (rev lst)))
  (g a (double rev-lst)))
```

into the alternative form

```
(g (* 2 pi)
   (append (rev lst)
           (rev lst)))
```

The main virtue of the proof of Figure 2.9 over that of Figure 2.8 is brevity, which is also the main advantage of the second piece of Scheme code over the first. The disadvantage in both cases is decreased modularity, as the reasoning behind the lemma M (or the computation behind `double`) is not abstracted away and separated into a distinct package. Concerning readability, we would argue that the proof in Figure 2.8 is more structured and lucid, although others could argue that the succinctness of the proof in Figure 2.9 makes it more readable. It is a testament to the versatility of DPLs that they readily support such a rich variety of styles. Furthermore, the affinity that these styles bear to corresponding programming styles—an affinity largely stemming

```
pick-any e1, e2, e3
  assume P = e1 ≈ e2 + e3
    (!tran (!tran (!-cong P (!ref e3))
                  (!+-assoc e2 e3 e3)
           (!tran (!+cong (!ref e2) (!inv e3))
                  (!id e2)))
```

Figure 2.9: A more succinct DPL proof of 2.1.

from a symmetry between the semantics of method abstraction and application on the one hand and functional abstraction and application on the other—corroborates our contention that proof engineering in DPLs is very similar to software engineering in functional programming languages.

"Mathematics is the most exact science, and its conclusions are capable of absolute proof. But this is so only because mathematics does not attempt to draw absolute conclusions. All mathematical truths are relative, conditional."

Charles Steinmetz

"Euclid taught me that without assumptions there is no proof. Therefore, in any argument, examine the assumptions."

E. T. Bell

3 Fundamentals

In this chapter we introduce the basic ideas behind our approach in a piecemeal fashion, by showing how one might arrive at a DPL for propositional logic. Although the style is elementary and the pace slow, most of the key insights originate in this setting. For this reason we believe that even readers with an advanced background would benefit from reading this chapter.

3.1 Formalizing classical reasoning as a DPL

3.1.1 Arguments and the nature of deduction

A deduction is what one gives in order to prove that an argument is valid. An argument consists of two things: a set of propositions known as the *premises*, and a single proposition known as the *conclusion*. The conclusion is supposed to be a logical, necessary consequence of the premises. If that is indeed the case, we say that the argument is *logically valid*, or simply valid. Differently put, an argument is valid if it is impossible for the premises to be true without the conclusion also being true. Note that nothing is said here about the truth of the conclusion or about the truth of the premises in isolation. All we are saying is that *if* the premises are in fact true, then the conclusion must also, necessarily, be true. Accordingly, if we accept the premises of a valid argument then we are logically compelled to accept the conclusion as well. Here is an example of a valid argument:

Premise 1: *Either I am the Pope or the moon is made of cheese.*
Premise 2: *I am not the Pope.*

Conclusion: *The moon is made of cheese.*

The argument is valid because the conclusion cannot possibly be false *if* the premises are true. It is an instance of the following general valid argument "schema":

Premise 1: $P \lor Q$
Premise 2: $\neg P$

Conclusion: Q

If at least one of P and Q holds, and P does not hold, then Q must hold.

As the example illustrates, validity by itself does not make a good argument. A valid argument might still lead us to a false conclusion. We also need *veracity*. An argument is veracious if its premises are true.[1] That is clearly not the case with the above example, the first premise having a rather serious flaw: the disjunction of me being the Pope or the moon being made of cheese does not hold, as in fact neither is the case. Therefore, a good argument must be both valid and veracious. Veracity guarantees that we start off from true premises; validity guarantees that the conclusion we reach will also be true.

Although necessary, the combination of validity and veracity is still not always sufficient for a good argument. A third requirement is epistemological: the premises must not simply be true, but we must have adequate justification for believing them to be true. Otherwise they are themselves in need of argument. From a logical point of view, however, our only concern is validity, that is, the question of whether the conclusion follows from the premises. Veracity and epistemic merit are to be decided by specialists in the field of the argument. By contrast, validity is established exclusively through logical considerations, by way of deduction.

Now judging from our foregoing example the reader might well wonder whether deduction is really necessary for establishing the validity of an argument. Is it not obvious just by immediate inspection whether or not the conclusion is logically entailed by the premises? In the case of the said example, the answer is yes, it is obvious, and no proof of any kind is really needed. But for most arguments deciding validity is far from trivial. As an example, consider the argument whose premises and conclusion are the axioms of number theory and Fermat's theorem, respectively; or the axioms of

[1] In traditional logic, a valid argument with true premises is called *sound*. We instead separate the truth of the premises from the issue of validity altogether, and reserve the term "sound" for more a technical use in connection with inference rules and deductions (which is more in line with contemporary practice in mathematical logic). Hence our use of the term "veracity".

ZF set theory and the continuum hypothesis; or some indubitable logical truths and the existence or non-existence of God; and so on. The validity or invalidity of such arguments is anything but obvious. *Demonstration* is necessary; the conclusion must be deductively derived from the premises. In fact we need not look at exotic examples to appreciate this point. Consider a much simpler argument:[2]

Premise 1: *Students who work hard get good grades.*
Premise 2: *Students who do not work hard enjoy college.*
Premise 3: *Students who do not get good grades do not enjoy college.*

Conclusion: *All students get good grades.*

The argument is valid, though this is not easily realized at first glance. Here is the proof: Pick any student S. Now either S works hard or not. If he does, then he gets good grades (by the first premise). If he does not, then he enjoys college (by the second premise). But the third premise implies that students who enjoy college must get good grades—for otherwise they would not enjoy college. Hence if S does not work hard he must get good grades. In any case, S gets good grades. Q.E.D.

We should do well to realize, however, that the conceptual complexity of subtle arguments is an empirical issue of psychology, having nothing to do whatsoever with the relation of logical consequence. That relation either obtains between the premises and the conclusion or it does not. The question is how easily we can come to discover which is the case. Fermat's theorem follows from the axioms of number theory just as necessarily as the conclusion $1 = 0$ follows from $0 = 1$. That is, the statement "if $0 = 1$ and $=$ is a symmetric relation then $1 = 0$" is just as much of an analytic truth as the statement "If the axioms of number theory and elementary logic hold then Fermat's theorem holds". Metaphysically, both statements are equally trivial truths. They could not possibly be false. Were we omnipotent beings we would immediately "see" that Fermat's theorem follows from the basic assumptions of number theory just as we "see" that the conclusion $1 = 0$ follows from $0 = 1$ and the usual conception of equality. But we are not omnipotent, and that is why deduction is indispensable. The logical links between the premises and the conclusion might be years of light apart, a distance which a human mind could not possibly perceive with one mental glance. A deduction will patiently guide us from the premises to the conclusion by putting all the links together in a step-by-step fashion that humans can follow.

You might have noticed that we have glossed over the notion of "logical consequence", or "follows logically from". We have been taking it for granted, without bothering to specify exactly when a proposition P is to count as a logical consequence of a set of propositions β (symbolically written as $\beta \models P$). We have explicated the

[2]This is a modification of an example of Kalish and Montague [41].

notion somewhat by saying that $\beta \models P$ holds if it is impossible for the propositions in β to be true without P also being true, but this will clearly not do as a formal analysis since it presupposes the concepts of truth and possibility. In fact a perfectly precise definition of logical consequence, resting on the notion of *interpretation*, is possible, and represents a major achievement of modern formal logic. But we will not go into the details here. We will assume that the reader has come across such a definition before. If not, an intuitive understanding of these ideas will suffice for our present purposes.

3.1.2 Inference rules and the general form of deductions

To sum up, a deduction is a chain of inferences adduced for the purpose of proving that an argument is valid.[3] The deduction takes the premises of the argument for granted, i.e., it treats them as working *assumptions* that hold by supposition. In DPL parlance, the set of assumptions that are taken as given is called the *assumption base*. The deduction proceeds in a piecemeal fashion through the application of *inference rules*. A sequence of steps are made, each of which establishes some intermediate conclusion by applying an inference rule to some of the assumptions and/or previously generated intermediate conclusions, until we finally reach the desired conslusion. Thus we may schematically depict the general skeleton of a deduction as follows:

$$\begin{array}{ccc} \textit{Line} & \textit{Conclusion} & \textit{Justification} \\ 1. & P_1 & \cdots \\ 2. & P_2 & \cdots \\ \vdots & \vdots & \vdots \\ n. & P_n & \cdots \end{array} \qquad \text{(I)}$$

Here each P_i is either an assumption (i.e., a member of the assumption base), or an intermediate conclusion; P_n is the final desired conclusion. The justification of each P_i typically consists of an application of an inference rule to some previous intermediate conclusions and/or assumptions. A typical inference rule is "left and-elimination": given a conjunction $A \wedge B$, we infer A. Another one with a more time-honored name is Modus Ponens: given the conditional $P \Rightarrow Q$ and P, we infer Q. Thus the justification for P_4, for instance, might read "Modus Ponens on P_2 and P_3". Of course if P_i is an assumption then it needs no justification; it holds by supposition.

A requirement of paramount importance is that the inference rules must be *sound*, i.e., they must be such that if their arguments are true then the result must also

[3] This linear view of a deduction as a "chain" will be seen to be inadequate, but it is a good first approximation for our purposes.

be true. A deduction can be accepted as conclusive evidence for the validity of an argument if and only if it uses sound rules. If it does then, on the assumption that the premises hold, a simple induction will show that every P_i holds too, including the desired conclusion P_n. For the basis step, P_1 must be true because it is either a premise or the result of applying a rule to some premises; and since the premises are assumed to be true and the rule is sound, the conclusion P_1 must be true. For the inductive step, consider proposition P_{i+1}, and assume that all preceding propositions are true. Then because P_{i+1} is derived by applying a sound rule to true propositions, it must itself be true. Hence the assumption that the premises are true implies that all propositions $P_1, \ldots, P_n$ in the deduction are true, including P_n; which is to say precisely that the argument is valid: if the premises are true, so is the conclusion.

But if an unsound rule is used, say on line i, then we have no reason for accepting P_i, because, by definition, the result that is generated by an unsound rule may well be false even if the arguments to the rule are true. And if P_i is false and is subsequently used to derive other propositions that eventually lead to the conclusion P_n, we clearly cannot consider the latter to be established, even on the assumption that the premises hold.

Now the cognitive value of a deduction stems from the simplicity of the primitive inference rules. Imagine a deduction system with a primitive inference rule that allowed us to derive Fermat's theorem from the basic assumptions of number theory. Then we could "prove" Fermat's theorem with an one-line deduction: the application of this rule to the axioms of number theory. Clearly, such a proof would be unacceptable.[4] The soundness of an inference rule must be readily intelligible, if not patently obvious; the rule must produce a conclusion that *clearly* follows from the arguments.

Note that each line in a deduction may be thought of as a mini argument by itself, for the application of an inference rule may be seen as an argument: the premises are the arguments to the rule, and the conclusion is the result that the rule produces. For instance, suppose that the third line obtains the conclusion B by applying Modus Ponens to some previous propositions $A \Rightarrow B$ and B. This can be thought of as the argument

$$\begin{array}{l}\text{Premise 1: } A \Rightarrow B \\ \text{Premise 2: } A \\ \hline \text{Conclusion: } B\end{array}$$

[4]Of course once Fermat's theorem *has already been proven* then it might be used as a lemma, i.e., as a *derived inference rule*. However, this would simply serve as a convenient abbreviation for the long-winded proof that we have already discovered. Every proof using derived inference rules can be expanded to a proof that uses only primitive inference rules, and those rules must be short and simple.

This is what a deduction does then: it breaks up a potentially intricate argument into a series of simple little arguments, the conclusion of each serving as a premise to some subsequent little argument. If each of the little arguments is self-evidently valid, and if every one of its premises is either a premise of the original argument or the conclusion of some previous "little argument", then by induction we may conclude that the final result is a logical consequence of the premises. The requirement that each little argument be valid amounts to requiring every inference rule to be sound; the requirement that each little argument be *self-evidently* valid amounts to requiring the soundness of every inference rule to be readily apprehensible.

3.1.3 Proof checking (linear case)

Let us now turn our attention to the task of *checking* a deduction—verifying that it soundly produces some particular conclusion. It is well-known that this problem is efficiently solvable, so in that respect the following discussion will not be saying anything new. Our purpose is simply to formulate and solve the problem in a way that will provide a smooth introduction to the main ideas behind our approach. In general terms the problem can be stated as follows. We are given:

1. An initial set of premises β—an initial *assumption base*, in DPL lingo. The elements of β will serve as working assumptions; they hold by supposition.

2. A deduction D of the form (I).

The question is: Does D successfully derive a conclusion that is logically entailed by β? If yes, what is that conclusion?

If we have a mechanical way of solving this problem then we can use it to determine whether a given deduction successfully produces a conclusion at all, or whether it makes an error of some kind instead. And, more importantly, if it does produce a conclusion, whether that is the *desired* conclusion (with respect to some given argument). So if someone presents us with a deduction D which he claims as proof for a certain argument with a set of premises β and conclusion P, we can settle his claim by using the algorithm.

Before we continue the reader should observe that our formulation of the problem has separated the assumption base from the deduction. This separation runs counter to most existing treatments of the subject, where a deduction is always hardwired to a specific set of premises. By sequestering the two we become free to speak of the meaning of a deduction *relative to an arbitrary set of premises*, and to check a deduction with respect to any given assumption base. Of course insofar we have constructed the deduction in order to establish a particular argument, the *intended* initial assumption base will consist of that argument's premises. But the insight behind DPLs is that, in general, the meaning of a deduction should be a function over assumption bases; it ought to be determined *relative* to a given assumption base. It turns out that this viewpoint pays handsome conceptual dividends and paves the way for some very elegant formal semantics. In some respects it is similar to the Tarskian conception of the meaning of a sentence as being *relative* to a given interpretation of the sentence's symbols. Of course we tend to think of the meaning of a sentence as fixed, because we usually have in mind a specific interpretation of the underlying vocabulary. But it is only by abstracting over all possible interpretations that we are able to arrive at the right formal semantics for satisfaction; from there all the other pieces (logical consequence and equivalence, validity, truth, etc.) fall together. In a sense, these advantages are replicated in the theory of DPLs. For instance, we become able to formally define (observational) equivalence for two deductions D_1 and D_2 by quantifying over assumption bases: $D_1 \approx D_2$ iff for all β, the meaning of D_1 relative to β is identical to the meaning of D_2 relative to β.

Let us continue with the checking problem. We will assume that there is a finite number of sound inference rules; that each rule takes a fixed number of propositions as inputs and produces a unique proposition as a result; that we can effectively determine whether a given list of propositions constitutes legitimate input to some rule; and if so, that we can effectively generate the rule's output for that particular input. These requirements are trivially satisfied by all common inference rules. Modus Ponens, for example, takes two propositions P_1 and P_2 as "inputs"; these are considered legitimate iff they are of the form $P \Rightarrow Q$ and Q, respectively; and in that case the resulting "output" is the proposition Q. Finally, we will assume that every entry in the "Justification" column of a deduction is either of the form *Rule* $P_1, \ldots, P_k$, signifying the application of *Rule* to the argument list $P_1, \ldots, P_k$; or some label (such as "Assumption") signifying that the conclusion of that line is a premise (i.e., in the assumption base).

The deduction-checking algorithm consists of one simple loop: we visit each line $i = 1, \ldots, n$ in turn. If the conclusion P_i is claimed as a premise, we check to see whether it is an element of β; if it is not, we report an error, otherwise we continue with the next line. On the other hand, if P_i is obtained by applying a rule R to a list of propositions $Q_1, \ldots, Q_k$, we check to make sure that

(a) the list $Q_1, \ldots, Q_k$ is appropriate input to the rule R, and that the conclusion P_i is indeed the result of applying R to $Q_1, \ldots, Q_k$; and that

(b) each Q_j, $j = 1, \ldots, k$, is either a premise (a member of β) or one of the previous $i - 1$ intermediate conclusions, $P_1, \ldots, P_{i-1}$.

If either condition fails, we report an error; otherwise we move on to the next line. If we finish all n lines without an error, then we announce that the deduction does produce a conclusion, namely, P_n: the conclusion of the last line. In view of the soundness of the inference rules, our inspection guarantees that the conclusion P_n follows logically from the premises, i.e., that $\beta \models P_n$. This follows by induction on n, as we explained earlier: we have $\beta \models P_i$ for every $i = 1, \ldots, n$.

If we make the simplifying assumption that the basic "units" of work in this algorithm are steps (a) and (b) above, then the worst-case complexity is $O(n^2)$; in particular, $\frac{n(n-1)}{2}$. The $i-1$ part of step (b) is responsible for the quadratic factor: for each argument Q_j, we might have to check all previous propositions to see whether Q_j is one of them. A more efficient way to check the deduction suggests itself: after P_i has been successfully verified, *add it to the assumption base* and then continue with the next line. Thus each P_i that has been successfully verified becomes a new premise, a new working assumption; more properly, it becomes a lemma. Step (b) is thus reduced to checking that each argument to the rule is in the *current* assumption base. It should be clear, again by inductive reasoning, that the modification is sound.

This "optimization" capitalizes on the aforementioned separation between the assumption base and the deduction, and its importance extends well beyond efficiency. The main benefit is conceptual: it helps to bring out the recursive structure and compositionality of deductions. We tend to think of a deduction as a block, but in fact it is more profitably viewed as a complex object that is *recursively* built up from simpler deductions through composition. Specifically, the original deduction D (the input to the algorithm) can be seen as a sequence of lines

$$D = L_1; \ldots; L_n$$

where the various lines are joined together by a composition operator ";". Now *split* this deduction at any point $i \in \{1, \ldots, n\}$. The key observation is that the remaining suffix $L_{i+1}, \ldots, L_n$ is itself a deduction, call it D_i. What is the relationship between D and D_i? Simple: D_i is sound with respect to $\beta \cup \{P_1, \ldots, P_i\}$ whenever D is sound with respect to β, where $P_1, \ldots, P_i$ are the conclusions of lines $1, \ldots, i$, respectively.

3.1.4 Arriving at a formal syntax and semantics

Let us make these intuitions more precise. First we notice that it is possible to give a recursive description of the structure of a deduction as follows: a deduction is either an application of an inference rule to a list of propositions; or a single proposition by itself (claimed as a premise); or a non-empty sequence of deductions. Let us call the first two kinds of deductions *primitive*, and the third kind of deductions *composite*;

and let us use the letter M for primitive deductions and the letter D for composite deductions. We thus arrive at the following *abstract syntax*:

$$D \;\; ::= \;\; M \;\mid\; M\,;D \tag{3.1}$$

where

$$M \;\; ::= \;\; P \mid \mathit{Rule} \;\; P_1, \ldots, P_n \tag{3.2}$$

Now it turns out there is a very elegant way to attach a semantics to this syntax so that a deduction D is meaningful iff it is sound; and if it is meaningful, its meaning (denotation) is the conclusion it produces. Theoretically, this is a very *natural* formalization of deduction. Practically, it has the useful consequence that evaluating a given deduction—i.e., obtaining its meaning—results either in error or in the conclusion produced by it. Thus evaluation becomes tantamount to proof-checking.

Let us work out a full "toy" formal semantics as an illustration. It will simplify the presentation to assume that there are only two inference rules: modus ponens (from $P \Rightarrow Q$ and P infer Q), and left and-elimination (from $P \wedge Q$ infer P). Everything we do can be straightforwardly extended to any finite number of rules; no new ideas are involved in such an extension. Thus we complete the abstract syntax above with the clause

$$\mathit{Rule} \;\; ::= \;\; \textbf{modus-ponens} \;\mid\; \textbf{left-and}$$

The style of our formal semantics will be denotational. Let *Ded* be the syntactic category of deductions, as generated by clause 3.1, and let **PrimDed** be the syntactic category of primitive deductions, as generated by clause 3.2. Further, let $\mathcal{B}$ be the domain of all assumption bases; let **Prop** be the set of all propositions (this does not need to be precisely defined for our purposes); and let *error* be some special token, distinct from all propositions. We give two meaning functions $\mathcal{M}$ and $\mathcal{D}$ with (curried) signatures

$$\mathcal{M}: \; \mathbf{PrimDed} \; \rightarrow \; \mathcal{B} \; \rightarrow \; \mathbf{Prop} \cup \{\mathit{error}\}$$

and

$$\mathcal{D}: \; \mathit{Ded} \; \rightarrow \; \mathcal{B} \; \rightarrow \; \mathbf{Prop} \cup \{\mathit{error}\}$$

Thus, intuitively, the meaning function $\mathcal{D}$ takes a deduction D and an assumption base β and produces either a proposition P or *error*. In more proper curried terms: $\mathcal{D}$ maps a given deduction to a function that takes an assumption base and gives either a proposition or *error*. The signature of $\mathcal{M}$ is likewise read. Since termination is guaranteed, no bottom elements are necessary.

We define $\mathcal{M}$ and $\mathcal{D}$ with a number of clauses, one for each form of M and D. As usual, recursion at the syntactic level is reflected by recursion at the semantic level. The clauses are shown in Fig. 3.1. To keep the equations clean, we do not explicitly

$$
\begin{array}{rcl}
\mathcal{M}[\![P]\!]\ \beta \cup \{P\} & = & P \\
\mathcal{M}[\![\textbf{left-and}\ P \wedge Q]\!]\ \beta \cup \{P \wedge Q\} & = & P \\
\mathcal{M}[\![\textbf{modus-ponens}\ P \Rightarrow Q, P]\!]\ \beta \cup \{P \Rightarrow Q, P\} & = & Q \\
\mathcal{D}\,[\![M]\!]\ \beta & = & \mathcal{M}[\![M]\!]\ \beta \\
\mathcal{D}\,[\![M; D]\!]\ \beta & = & \mathcal{D}\,[\![D]\!]\ \beta \cup \{\mathcal{M}[\![M]\!]\ \beta\}
\end{array}
$$

Figure 3.1: Denotational semantics of simple deductions.

enumerate all the cases that lead to error. Rather, we tacitly assume that any "input" to the functions $\mathcal{M}$ and $\mathcal{D}$ that does not match the left-hand side of one of the equations produces *error*. Thus, for example, we have

$$\mathcal{M}[\textbf{modus-ponens}\ P \wedge Q]\ \beta = \mathit{error}$$

and $\mathcal{M}[P]\ \beta = \mathit{error}$ for any β that does not contain P. Moreover, we assume that errors are propagated in the obvious way; for instance, in the right-hand side of the equation for $\mathcal{D}$, if the value of the sub-expression $\mathcal{M}[M]\ \beta$ is *error*, then the value of the entire expression is *error*.

The equations of Fig. 3.1 can be directly transcribed into an evaluation algorithm. That algorithm will be *precisely* the deduction checking algorithm we discussed above. This means that the semantics of the language capture exactly what it means for a deduction to be sound, and reinforces the slogan

$$\mathit{Evaluation} = \mathit{Proof\ Checking}.$$

Fig. 3.2 offers an alternative presentation of the semantic equations that makes explicit the dependence of a deduction's meaning on an assumption base. Thus the equations of Fig. 3.2 make it clear that the meaning of a deduction should be understood *relative* to a given set of premises.

3.1.5 Non-linear case: assumption scope

The linear view of deductions we have been painting is not accurate. Oftentimes one of the intermediate conclusions P_i established in the course of a deduction is derived not by the straightforward application of some inference rule but rather by provisionally postulating some proposition Q_1 (the "hypothesis") and then showing that if the latter holds then some other proposition Q_2 holds as well (the hypothetical conclusion). The

$$
\begin{array}{rcl}
\mathcal{M}[\![P]\!] & = & \lambda\,\beta.\,P \in \beta\,? \rightarrow P, \mathit{error} \\
\mathcal{M}[\![\textbf{left-and}\ P \wedge Q]\!] & = & \lambda\,\beta.\,P \wedge Q \in \beta\,? \rightarrow P, \mathit{error} \\
\mathcal{M}[\![\textbf{modus-ponens}\ P \Rightarrow Q, P]\!] & = & \lambda\,\beta.\,\{P \Rightarrow Q, P\} \subseteq \beta\,? \rightarrow Q, \mathit{error} \\
\mathcal{D}\,[\![M]\!] & = & \lambda\,\beta.\,\mathcal{M}[\![M]\!]\,\beta \\
\mathcal{D}\,[\![M; D]\!] & = & \lambda\,\beta.\,\mathcal{D}\,[\![D]\!]\,\beta \cup \{\mathcal{M}[\![M]\!]\,\beta\}
\end{array}
$$

Figure 3.2: Alternative formulation of the denotational semantics.

idea is that the derivation of Q_2 from Q_1 is sound justification for inferring P_i. We call these *hypothetical deductions*. Hypothetical deductions arise in two very important cases:

- in establishing conditionals, i.e., propositions of the form $P \Rightarrow Q$; and
- in establishing negations, i.e., propositions of the form $\neg P$.

We discuss each in turn and show how each can be accomodated in our model by appropriately extending the abstract syntax and denotational semantics we have developed for linear deductions.

The customary method for establishing a conditional $P \Rightarrow Q$ is this: we assume P and proceed to infer Q; upon deriving Q (with the aid of the hypothesis P), we derive the implication $P \Rightarrow Q$ and tacitly retract P from the current set of working assumptions. As a very simple example, consider proving the conditional

$$\mathit{Even}(n) \Rightarrow \mathit{Even}(n+2)$$

where n ranges over the integers and the predicate *Even* is defined as

$$\mathit{Even}(n) \equiv_{def} (\exists k) n = 2k.$$

We reason as follows:

> Assume that n is even. Then, by definition, we must have $n = 2k$ for some number k. Therefore, $n+2 = 2k+2 = 2(k+1)$, which proves $\mathit{Even}(n+2)$. Q.E.D.

This mode of reasoning can be readily analyzed in terms of assumption bases. The general case is as follows: we are proceeding along a deduction linearly, by sequentially applying inference rules, and then we get to a point where we need to establish an intermediate conclusion of the form $P \Rightarrow Q$. Now there may be a way to get this conclusion the simple way, by applying an inference rule. For instance, one of our previous intermediate conclusions might be of the form $Q_3 \wedge (P \Rightarrow Q)$. In that case a simple application of and-elimination will suffice. But if we cannot derive $P \Rightarrow Q$ through a rule, then what we commonly do, as we saw above, is add the hypothesis P to our current assumption base β and construct a deduction D' that derives Q from this enlarged assumption base, $\beta \cup \{P\}$. Upon deriving Q, we conclude $P \Rightarrow Q$ and continue along with β as our assumption base again, having thus "discharged" the hypothesis P. To prove that this method is sound, we need to show that β logically implies the conclusion produced by the method, namely $P \Rightarrow Q$. But this can be easily done inductively: Assuming that D' is sound, we have $\beta \cup \{P\} \models Q$, which in turn implies $\beta \models P \Rightarrow Q$, by the definition of $\models$ and the semantics of the connective $\Rightarrow$. Note that D' might itself contain other hypothetical deductions, which may themselves contain hypothetical deductions, and so on.

We have thus given a natural explanation of hypothetical reasoning in terms of the recursive structure of deductions (notice in particular the role played by the *subdeduction* D' that derives Q) and the semantic abstraction of assumption bases. In this light we see that the assumption base, far from being static, grows and shrinks dynamically in the course of a deduction. The expansions and contractions correspond precisely to assumption introductions and discharges and match up symmetrically. Now this is paradigmatic context-free behavior; it is intuitively reminiscent of the epitomy of context free languages: $a^n b^n$ (or $(^n)^n$, if you will). Indeed, it turns out that context-free block structure is ideal for capturing this behavior syntactically; and this is one of the greatest advantages of DPLs. Semantically, the intuitive manipulation of assumption bases we described above (insert the hypothesis upon entering the body of the hypothetical deduction, retract it at the end) confers just the right meaning to the corresponding block-structured syntax. Recursion carries this harmonious arrangement to arbitrarily deep levels of nesting. Thus it comes about that the introduction and discharge of assumptions, traditionally one of the most troublesome aspects of formal deduction, is taken care of automatically—and elegantly—by the syntax and semantics of the language. Moreover, this results in an accurate formal model of the way in which humans perform hypothetical reasoning in practice.

To make things precise, let us account for hypothetical deductions by extending the abstract syntax 3.1 as follows:

$$D \ ::= \ M \ \mid \ D_1 \,; D_2 \mid \textbf{assume } P \textbf{ in } D \tag{3.3}$$

where primitive deductions M remain as before. Intuitively, this grammar says that a deduction is either

- a primitive deduction (namely, a claim P or an application of an inference rule to a list of propositions); or
- a composite deduction $D_1; D_2$, whereby D_1 is followed by D_2; or
- a hypothetical deduction of the form **assume** P **in** D, consisting of an assumption P (the *hypothesis*), and a *body* D.

Note that the grammar 3.3 is now ambiguous. For instance, it is not clear whether a deduction of the form **assume** P **in** $D_1; D_2$ represents a hypothetical deduction with body $D_1; D_2$, or a composite deduction with **assume** P **in** D_1 and D_2 as first and second components, respectively. We will remove any such ambiguities by using **begin-end** pairs.

The denotational semantics for the new language are more or less the same as before. We need only add a new clause giving the meaning of hypothetical deductions:

$$\mathcal{D}\,[\![\textbf{assume } P \textbf{ in } D]\!]\,\beta = P \Rightarrow \mathcal{D}\,[\![D]\!]\,\beta \cup \{P\} \tag{3.4}$$

and slightly modify the composition rule to account for the new syntax ($D_1; D_2$ rather than $M; D$):

$$\mathcal{D}\,[\![D_1; D_2]\!]\,\beta = \mathcal{D}\,[\![D_2]\!]\,\beta \cup \{\mathcal{D}[\![D_1]\!]\,\beta\} \tag{3.5}$$

The important point is that in equation 3.4, the deduction D—the body of the hypothetical deduction—is evaluated in the assumption base β *augmented with* P. Thus the meaning of the entire **assume** P **in** D deduction with respect to a given β is the conditional whose antecedent is the hypothesis P and whose consequent is the meaning of the body D with respect to $\beta \cup \{P\}$.

As another example, suppose we have two more primitive inference rules, a unary rule **right-and**, acting as the obvious analogue of **left-and**, and a binary rule **both**, which takes two propositions P and Q in the current assumption base and returns the conjunction $P \wedge Q$ (i.e., **both** is an *introduction rule* for the connective $\wedge$). Hence the denotational clause for **both** is:

$$\mathcal{M}\,[\![\textbf{both}\, P, Q]\!]\,\beta \cup \{P, Q\} = P \wedge Q.$$

Then here is a deduction that proves the conditional $P \wedge Q \Rightarrow Q \wedge P$, for arbitrary P and Q:

assume $P \wedge Q$ **in**
  begin
    right-and $P \wedge Q$;
    left-and $P \wedge Q$;
    both Q, P;
  end;

Call this deduction D. The diligent reader will formally verify that the denotational semantics we have given ensure the equality

$$\mathcal{D}\,[\![D]\!]\,\emptyset = P \wedge Q \Rightarrow Q \wedge P$$

for any choice of P and Q.

The second kind of hypothetical reasoning arises in connection with *reasoning by contradiction*. Suppose we wish to establish $\neg P$, the negation of some proposition P. This is often done by showing P to be inconsistent with our current premises: we add P to our premises and derive a contradiction. Upon establishing the contradiction we conclude $\neg P$ and tacitly retract P from the current set of working assumptions. As an example, consider the following argument:

Premise 1: *Mathematicians are intelligent.*
Premise 2: *Intelligent people do not get arrested.*
Premise 3: *Alfred is a mathematician.*
Premise 4: *The murderer was arrested.*

Conclusion: *The murderer was not Alfred.*

Here is a deduction showing that the conclusion follows from the premises: Suppose, by way of contradiction, that the murderer is Alfred. Then by the fourth premise we conclude that Alfred was arrested. But Alfred is a mathematician (third premise), hence by the first premise we may conclude that Alfred is intelligent. Accordingly, by the second premise we infer that Alfred was not arrested. But now we have a contradiction: we have concluded both that Alfred has been arrested and that he has not. Therefore, our assumption that Alfred is the murderer is inconsistent with our premises and must be rejected. We conclude that the murderer was not Alfred.

The reader at this point might be able to guess how our model can capture such reasoning. First we augment the abstract syntax of deductions with one extra production:

$$D \ ::= \ M \ | \ D_1 \,;D_2 \mid \textbf{assume}\ P\ \textbf{in}\ D \mid \textbf{suppose-absurd}\ P\ \textbf{in}\ D \tag{3.6}$$

Informally, a deduction of the form **suppose-absurd** P **in** D is evaluated as follows (in the context of a given β): we insert P in β and proceed to evaluate the body D. If the conclusion of D in $\beta \cup \{P\}$ is an absurdity (namely, the propositional constant **false**), then we return the negation $\neg P$; otherwise we report an error. More precisely, we assume we have a binary primitive inference rule **absurd** that takes two propositions of the form P and $\neg P$ as inputs and returns the proposition **false**; any other input to **absurd** results in error. Thus we have

$$\mathcal{M}\,[\![\textbf{absurd}\ P, \neg P]\!]\,\beta = \textbf{false}.$$

Then the formal semantics of the **suppose-absurd** construct can be defined as follows:

$$\mathcal{D}\,[\![\textbf{suppose-absurd}\;\; P\;\; \textbf{in}\; D]\!]\; \beta = \mathcal{D}\,[\![D]\!]\; \beta \cup \{P\} = \textbf{false}\,? \rightarrow \neg P,\; \mathit{error}. \tag{3.7}$$

With these semantics, here is a deduction that derives the conditional $P \Rightarrow \neg\neg P$:

$$
\begin{array}{l}
\textbf{assume}\;\; P\; \textbf{in} \\
\quad \textbf{suppose-absurd}\;\;\; \neg P\; \textbf{in} \\
\qquad \textbf{absurd}\;\;\; P, \neg P
\end{array}
$$

In the sequel we will see that the ideas we have sketched here can be naturally extended to full first-order reasoning, and indeed to various other logics.

"He does it with a better grace, but I do it more natural."
William Shakespeare, Twelfth Night

4 Classical Natural Deduction

In this chapter we introduce $\mathcal{CND}$, a DPL for classical propositional logic. We present its syntax and semantics, develop its theory, including several notions of formal proof equivalence and the relations amongst them, give several examples of proofs written in it, and show that it is sound and complete. We close the chapter by studying proof composition in a general setting, and by discussing what would be involved in augmenting $\mathcal{CND}$ with an abstraction mechanism. $\mathcal{CND}$ will be extended to predicate logic in Chapter 6.

4.1 Syntax

We use the letters P, Q, $R, \ldots$, to designate arbitrary *propositions*. Propositions are built from the following abstract grammar:

$$P ::= A \mid \mathbf{true} \mid \mathbf{false} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q$$

where A ranges over a countable set of atomic propositions ("atoms") which we need not specify in detail. The letters A, B, and C will be used as typical atoms. Parentheses will be used as necessary to disambiguate parsing.

The *deductions* of $\mathcal{CND}$ have the following abstract syntax:

$$D ::= P \mid \textit{Prim-Rule}\ P_1, \ldots, P_n \mid \mathbf{assume}\ P\ \mathbf{in}\ D \mid D_1; D_2$$

Prim-Rule	::=	**modus-ponens**	($\Rightarrow$-elimination)
	|	**modus-tollens**	($\neg$-introduction)
	|	**double-negation**	($\neg$-elimination)
	|	**both**	($\wedge$-introduction)
	|	**left-and**	($\wedge$-elimination)
	|	**right-and**	($\wedge$-elimination)
	|	**left-either**	($\vee$ -introduction)
	|	**right-either**	($\vee$ -introduction)
	|	**constructive-dilemma**	($\vee$ -elimination)
	|	**equivalence**	($\Leftrightarrow$ -introduction)
	|	**left-iff**	($\Leftrightarrow$ -elimination)
	|	**right-iff**	($\Leftrightarrow$ -elimination)
	|	**absurd**	(**false**-introduction)

Figure 4.1: Primitive inference rules

where *Prim-Rule* ranges over a collection of primitive inference rules such as Modus Ponens. The decision of exactly which rules to choose as primitives has bearing mostly on the metatheoretical properties of the language (soundness and completeness in particular). For instance, if we take as a primitive rule one that derives $P \wedge Q$ from P, we will clearly have an unsound language; and if we leave out certain rules such as $\Leftrightarrow$-elimination, then we will have an incomplete language. The rules shown in Figure 4.1 will allow for a sound and complete natural deduction system, as we will prove later. We will occasionally write **mp**, **mt**, **dn**, and **cd** as abbreviations for **modus-ponens**, **modus-tollens**, **double-negation**, and **constructive-dilemma**, respectively.

The reader will notice one omission from Figure 4.1: an introduction rule for $\Rightarrow$. This has traditionally been the most troublesome spot for classical deduction systems (see Section 7.1). As we will see shortly, in $\mathcal{CND}$ conditionals are introduced via the language construct **assume**, in a manner that avoids most of the usual problems. Moreover, although we have listed **modus-tollens** as an introduction rule for negation, in practice negations in $\mathcal{CND}$ are usually introduced via deductions of the form

$$\textbf{suppose-absurd } P \textbf{ in } D \tag{4.1}$$

that perform reasoning by contradiction. Such deductions are not taken as primitive because their intended behavior is expressible in terms of **assume**, primitive inference rules, and composition. In particular, we define 4.1 as an abbreviation for the deduction

assume P **in** D;
$\neg$**false**;
modus-tollens $P \Rightarrow$ **false**, $\neg$**false**

Part (b) of Lemma 4.2 below will show that this desugaring gives the expected semantics (as discussed in Chapter 3).

A deduction will be called a *claim* if it is of the form P;[1] a *primitive* (or *atomic*) *deduction* if it is of the form *Prim-Rule* $P_1, \ldots, P_n$; a *hypothetical* or *conditional deduction* if it is of the form **assume** P **in** D; and a *composition* (or *sequence*) if it is of the form $D_1; D_2$. In a hypothetical deduction of the above form, P and D are called the *hypothesis* and the *body* of the deduction, respectively. The body represents the *scope* of the corresponding hypothesis. A deduction will be called *non-trivial* iff it is not a claim (thus claims are viewed as trivial deductions). We define $SZ(D)$, the *size* of a deduction D, by structural recursion:

$$\begin{array}{rcl} SZ(P) & = & 1 \\ SZ(\textit{Prim-Rule}\ P_1, \ldots, P_n) & = & n+1 \\ SZ(\textbf{assume}\ P\ \textbf{in}\ D) & = & SZ(D)+1 \\ SZ(D_1; D_2) & = & SZ(D_1)+SZ(D_2) \end{array}$$

Note that the grammar we have given above specifies the abstract syntax of the language; it cannot serve as a concrete syntax because it is ambiguous. For instance, it is not clear whether

$$\textbf{assume}\ P \wedge Q\ \textbf{in}\ \textbf{true};\ \textbf{left-and}\ P \wedge Q$$

is a hypothetical deduction with the composition **true**; **left-and** $P \wedge Q$ as its body, or a composition consisting of a hypothetical deduction followed by a primitive application of **left-and**. We will use **begin-end** pairs or parentheses to resolve any such ambiguity. We also stipulate that the composition operator ; associates to the right, so that $D_1; D_2; D_3$ will stand for $D_1; (D_2; D_3)$. This is an important convention because we will prove later that, unlike its counterpart in imperative programming languages, the said operator is *not* associative (in a sense of associativity that will be made precise in Section 4.5).

Occasionally it is expedient to adopt a two-dimensional view of deductions, treating a given D explicitly as an abstract syntax tree and using a tree-like positional

[1] Claims need not form a separate syntactic category; they could be subsumed by primitive deductions by introducing a unary inference rule **claim** which returns its argument if the latter is in the assumption base and fails otherwise (in fact in the $\lambda\mu$-calculus that must be so by necessity, since a proposition by itself is an expression, not a deduction). We treat claims separately here for exposition purposes.

notation for identifying different parts of it (similar to that used for Herbrand terms, see Section A.1). In such a scheme the "position" of a part of D is represented by a (possibly empty) list of positive integers describing the path that one must traverse in order to get from the root of D to the part in question. This idea is made precise in Section B.1. Appendix B also contains definitions of a few other notions such as subdeductions, threads, and the relation of dominance. These concepts will not be needed in this chapter, but will be used in the next one.

4.2 Evaluation semantics

As we explained in the previous chapter, the purpose of a deduction is to establish a certain conclusion on the basis of some given assumptions known as *premises*. Accordingly, specifying the semantics of our language amounts to specifying exactly what conclusion—if any—a given deduction D is capable of producing relative to a given set of assumptions. This last phrase, "relative to a given set of assumptions", is central to our approach. Just as a program in a language such as Pascal or ML is always executed relative to a given *store*, which can be understood as a description of the contents of the computer's memory, a deduction in a denotational proof language such as $\mathcal{CND}$ is always evaluated relative to a given *assumption base*, which can be understood as a set of propositions that we are taking as "given", i.e., as premises. In what follows we will use the term "assumption base" synonymously with "finite set of propositions".[2] We will use the letter β to represent assumption bases, and the letters Φ and Ψ for arbitrary—possibly infinite—sets of propositions.

We will present the semantics of $\mathcal{CND}$ in the style of Kahn [40]. The evaluation judgments will be of the form $\beta \vdash D \rightsquigarrow P$, which may be read as follows: "In the context of β, D yields the conclusion P", or, more operationally, "from β we can prove that D derives P". The semantic clauses are partitioned into two groups: those dealing with compound deductions and claims, shown in Figure 4.2, and those dealing with primitive deductions, shown in Figure 4.3.

We say that the judgment $\beta \vdash D \rightsquigarrow P$ is *derivable* iff there exists a derivation of it, namely, a finite sequence of judgments

$$\beta_1 \vdash D_1 \rightsquigarrow P_1, \ldots, \beta_n \vdash D_n \rightsquigarrow P_n \tag{4.2}$$

[2]The use of finite sets is not essential; all of our subsequent definitions and results could also be obtained for infinite assumption bases. However, the use of finite assumption bases simplifies some technical issues, especially in the first-order setting. At any rate, for the purpose of mechanically evaluating deductions we could not consider arbitrary assumption bases anyway; we would have to restrict attention to recursive assumption bases (those with a computable decision problem), since we must be able to tell in a finite amount of time whether a given claim is a member of a certain assumption base. Working with finite assumption bases ensures this *a priori*, and facilitates implementation.

$$\frac{}{\beta \vdash \textbf{true} \leadsto \textbf{true}}\ [R_1] \qquad \frac{}{\beta \vdash \neg\textbf{false} \leadsto \neg\textbf{false}}\ [R_2]$$

$$\frac{}{\beta \cup \{P\} \vdash P \leadsto P}\ [R_3] \qquad \frac{\beta \cup \{P\} \vdash D \leadsto Q}{\beta \vdash \textbf{assume } P \textbf{ in } D \leadsto P \Rightarrow Q}\ [R_4]$$

$$\frac{\beta \vdash D_1 \leadsto P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \leadsto P_2}{\beta \vdash D_1; D_2 \leadsto P_2}\ [R_5]$$

Figure 4.2: The semantics of $\mathcal{CND}$.

such that $\beta_n = \beta, D_n = D, P_n = P$, and where each judgment $\beta_i \vdash D_i \leadsto P_i$ is either an instance of one of the axioms $[R_1]$, $[R_2]$, or $[R_3]$; or an instance of one of the axioms for primitive deductions shown in Figure 4.3; or else follows from previous judgments through $[R_4]$ or $[R_5]$. We indicate that $\beta \vdash D \leadsto P$ is derivable by writing $\vdash_{\mathcal{CND}} \beta \vdash D \leadsto P$, or simply $\beta \vdash D \leadsto P$. For an arbitrary set of propositions Φ, we write $\Phi \vdash_{\mathcal{CND}} P$ to mean that there is a deduction D and a subset $\beta \subseteq \Phi$ such that $\beta \vdash D \leadsto P$. If $\Phi \vdash_{\mathcal{CND}} P$ we say that P is *provable* (or "deducible", or "derivable") from Φ. Our first result can be proved by structural induction on D (it also follows as a corollary of Theorem 4.7 or Theorem 4.8 below):

Theorem 4.1 (Uniqueness) *If* $\beta_1 \vdash D \leadsto P_1$ *and* $\beta_2 \vdash D \leadsto P_2$ *then* $P_1 = P_2$.

The following captures the essence of hypothetical deductions and reasoning by contradiction (based on the desugaring of **suppose-absurd** given in the previous section):

Lemma 4.2 (a) *If* $\beta \cup \{P\} \vdash D \leadsto Q$ *then* $\beta \vdash \textbf{assume } P \textbf{ in } D \leadsto P \Rightarrow Q$*; and* (b) *if* $\beta \cup \{P\} \vdash D \leadsto \textbf{false}$ *then* $\beta \vdash \textbf{suppose-absurd } P \textbf{ in } D \leadsto \neg P$.

The reflexivity lemma below is a direct consequence of $[R_3]$. Monotonicity is immediate too. The dilution result also expresses a form of monotonicity, but it makes a different assertion from Lemma 4.4. The latter holds trivially, by the definition of $\Phi \vdash_{\mathcal{CND}} P$. By contrast, Lemma 4.5 makes a stronger statement: it says that if *a particular deduction* D yields a conclusion P in the context of some β, then that will remain the case even if we "dilute" β with any number of additional propositions. This can be proved by an induction on the length of the derivation of $\beta \vdash D \leadsto P$.

Lemma 4.3 (Reflexivity) $\Phi \vdash_{\mathcal{CND}} P$ *for all* $P \in \Phi$.

$$\beta \cup \{P \Rightarrow Q, P\} \vdash \textbf{modus-ponens}\ P \Rightarrow Q, P \leadsto Q$$
$$\beta \cup \{P \Rightarrow Q, \neg Q\} \vdash \textbf{modus-tollens}\ P \Rightarrow Q, \neg Q \leadsto \neg P$$
$$\beta \cup \{\neg\neg P\} \vdash \textbf{double-negation}\ \neg\neg P \leadsto P$$
$$\beta \cup \{P_1, P_2\} \vdash \textbf{both}\ P_1, P_2 \leadsto P_1 \wedge P_2$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{left-and}\ P_1 \wedge P_2 \leadsto P_1$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{right-and}\ P_1 \wedge P_2 \leadsto P_2$$
$$\beta \cup \{P_1\} \vdash \textbf{left-either}\ P_1, P_2 \leadsto P_1 \vee P_2$$
$$\beta \cup \{P_2\} \vdash \textbf{right-either}\ P_1, P_2 \leadsto P_1 \vee P_2$$
$$\beta \cup \{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\} \vdash \textbf{constructive-dilemma}\ P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q \leadsto Q$$
$$\beta \cup \{P_1 \Rightarrow P_2, P_2 \Rightarrow P_1\} \vdash \textbf{equivalence}\ P_1 \Rightarrow P_2, P_2 \Rightarrow P_1 \leadsto P_1 \Leftrightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{left-iff}\ P_1 \Leftrightarrow P_2 \leadsto P_1 \Rightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{right-iff}\ P_1 \Leftrightarrow P_2 \leadsto P_2 \Rightarrow P_1$$
$$\beta \cup \{P, \neg P\} \vdash \textbf{absurd}\ P, \neg P \leadsto \textbf{false}$$

Figure 4.3: Axioms for primitive deductions.

Lemma 4.4 (Monotonicity) *If* $\Phi \vdash_{\mathcal{CND}} P$ *then* $\Phi \cup \Psi \vdash_{\mathcal{CND}} P$.

Lemma 4.5 (Dilution) *If* $\beta \vdash D \leadsto P$ *then* $\beta \cup \beta' \vdash D \leadsto P$.

The "big-step" evaluation semantics we have given here offers clarity and generality by avoiding the specification of details, but as a result it is not sufficiently operational. In particular, it gives little insight as to when a judgment $\beta \vdash D \leadsto P$ *fails* to be derivable. A more algorithmic semantics is given by the interpeter *Eval*, shown in Figure 4.4 in pseudo-ML notation, which, owing to Theorem 4.7 below, may be seen as an implementation of the formal semantics. *Eval* uses an auxiliary function *do-prim-rule* which can be found in Section B.1. The call $Eval(D, \beta)$ evaluates D in the assumption base β, either resulting in a proposition P (the conclusion of D) or in *error*. Termination is assured:

Theorem 4.6 (Termination) *Eval always terminates. In particular, the recursion tree spawned by a call* $Eval(D, \beta)$ *has size* $\Theta(n)$*, where* n *is the size of* D*. Accordingly, a deduction is evaluated in time linear in its size.*

Proof: By structural induction on D. ■

The following result relates the interpreter to the formal semantics; it can be proved by induction on the structure of D:

$$
\begin{array}{l}
\mathit{Eval}(D, \beta) = \mathit{ev}(D) \\
\text{where} \\
\quad \mathit{ev}(P) = P \in \beta \cup \{\mathbf{true}, \neg\mathbf{false}\} \rightarrow P, \mathit{error} \\
\quad \mathit{ev}(\mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n) = \mathit{do\text{-}prim\text{-}rule}(\mathit{Prim\text{-}Rule}, [P_1, \ldots, P_n], \beta) \\
\quad \mathit{ev}(\mathbf{assume}\ P\ \mathbf{in}\ D) = \mathit{let}\ \ Q = \mathit{Eval}(D, \beta \cup \{P\}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad Q = \mathit{error} \rightarrow \mathit{error}, P \Rightarrow Q \\
\quad \mathit{ev}(D_1; D_2) = \mathit{let}\ \ P_1 = \mathit{Eval}(D_1, \beta) \\
\qquad\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad\qquad P_1 = \mathit{error} \rightarrow \mathit{error}, \mathit{Eval}(D_2, \beta \cup \{P_1\})
\end{array}
$$

Figure 4.4: An interpreter for $\mathcal{CND}$.

Theorem 4.7 (Correctness of *Eval*) *For all P and D,*

$$\beta \vdash D \leadsto P \ \text{ iff } \ \mathit{Eval}(D, \beta) = P.$$

Hence, by termination, $\mathit{Eval}(D, \beta) = \mathit{error}$ iff there is no P such that $\beta \vdash D \leadsto P$.

4.3 Basic $\mathcal{CND}$ theory

We will say that a deduction is *well-formed* iff every primitive subdeduction of it has one of the forms shown in Figure 4.3. Thus, loosely put, a deduction is well-formed iff the right number and kind of arguments are supplied to every application of a primitive rule, so that, for instance, there are no applications such as **modus-ponens** $A \wedge B, B$ or **left-and true**. Clearly, "type-checking" a deduction to make sure that it is well-formed is a trivial matter; in particular, it is not necessary to maintain and manipulate an assumption base, i.e., to keep track of intermediate conclusions, hypotheses, etc. In other words this is a purely syntactic concept—having nothing to do with assumption bases—and we could have introduced it in the previous section via a simple type system establishing judgments of the form $\vdash_W D$ ("D is well-formed"); we present such a type system in Appendix B.1. From here on we will only be concerned with well-formed deductions, unless we explicitly say otherwise.

The *conclusion* of a well-formed deduction D, denoted $\mathcal{C}(D)$, is defined by structural recursion: the conclusion of a claim is the claim itself; the conclusion of a primitive deduction can be defined by enumerating the various cases as in Figure 4.3 (e.g., the conclusion of a deduction of the form **modus-ponens** $P \Rightarrow Q, P$ is Q, the conclusion of a deduction of the form **double-negation** $\neg\neg P$ is P, etc.—consult the appendix for an exhaustive enumeration); the conclusion of a hypothetical deduction **assume** P **in** D' is $P \Rightarrow \mathcal{C}(D')$; and the conclusion of a sequence $D_1; D_2$ is the conclusion of D_2.

Clearly, this definition can be used as a recursive algorithm for computing $\mathcal{C}(D)$. Computing $\mathcal{C}(D)$ is quite different—much easier—than evaluating D. For example, if D is of the form $D_1, D_2, \ldots, D_{99}, D_{100}$, we can completely ignore the first ninety-nine deductions and simply compute $\mathcal{C}(D_{100})$, since, by definition, $\mathcal{C}(D) = \mathcal{C}(D_{100})$. The catch, of course, is that D might fail to establish its conclusion (in a given assumption base); evaluation is the only way to find out. However, it is easy to show (Theorem 4.8 below) that *if* $\beta \vdash D \leadsto P$, then $P = \mathcal{C}(D)$; thus, in terms of the interpreter, either $\mathit{Eval}(D, \beta) = \mathit{error}$ or $\mathit{Eval}(D, \beta) = \mathcal{C}(D)$, for any D and β. Loosely paraphrased: a deduction succeds iff it produces its professed conclusion.

Theorem 4.8 *If* $\beta \vdash D \leadsto P$ *then* $P = \mathcal{C}(D)$.

Proof: We will use strong induction on the length n of the derivation of the judgment $\beta \vdash D \leadsto P$. When $n = 1$ the judgment must be an instance of $[R_1]$, or $[R_2]$, or $[R_3]$, or one of the axioms for primitive deductions shown in Figure 4.3. In any case a straightforward inspection will show that the result holds. For the inductive step assume that $n > 1$, where $\beta \vdash D \leadsto P$ is the last (n^{th}) element of the derivation, and that the result holds for all judgments that can be derived in fewer than n steps. We proceed with a case analysis of the rule by which the judgment $\beta \vdash D \leadsto P$ is obtained:

$[R_4]$: In that case D is of the form **assume** P_1 **in** D' and we must have $P = P_1 \Rightarrow P_2$, where the judgment $\beta \cup \{P_1\} \vdash D' \leadsto P_2$ is derivable in fewer than n steps. Hence, by the inductive hypothesis,

$$P_2 = \mathcal{C}(D'). \tag{4.3}$$

Now $\mathcal{C}(D) = P_1 \Rightarrow \mathcal{C}(D')$, so, from 4.3, $\mathcal{C}(D) = P_1 \Rightarrow P_2$, i.e., $\mathcal{C}(D) = P$.

$[R_5]$: In that case D is of the form $D_1; D_2$ and $P = P_2$, where the judgments

$$\beta \vdash D_1 \leadsto P_1 \text{ and } \beta \cup \{P_1\} \vdash D_2 \leadsto P_2$$

are derivable in fewer than n steps. Inductively, $P_2 = \mathcal{C}(D_2)$. But $\mathcal{C}(D) = \mathcal{C}(D_2)$, hence $\mathcal{C}(D) = P_2 = P$.

This completes the case analysis and the induction. ■

Corollary 4.9 *For all* D *and* β*, either* $\mathit{Eval}(D, \beta) = \mathit{error}$ *or* $\mathit{Eval}(D, \beta) = \mathcal{C}(D)$.

$$
\begin{array}{rcll}
FA(\mathbf{true}) & = & FA(\neg\mathbf{false}) \;=\; \emptyset & (4.4)\\
FA(P) & = & \{P\} \;\; (\text{for } P \notin \{\mathbf{true}, \neg\mathbf{false}\}) & (4.5)\\
FA(\mathbf{left\text{-}either}\ P_1, P_2) & = & \{P_1\} & (4.6)\\
FA(\mathbf{right\text{-}either}\ P_1, P_2) & = & \{P_2\} & (4.7)\\
FA(\mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n) & = & \{P_1, \ldots, P_n\} & (4.8)\\
FA(\mathbf{assume}\ P\ \mathbf{in}\ D) & = & FA(D) - \{P\} & (4.9)\\
FA(D_1; D_2) & = & FA(D_1) \cup (FA(D_2) - \{\mathcal{C}(D_1)\}) & (4.10)
\end{array}
$$

Figure 4.5: Definition of $FA(D)$.

Next we define the set of *free* (or *strict*) *assumptions* of a deduction D, denoted $FA(D)$, as shown in Figure 4.5, where we assume that

$$\mathit{Prim\text{-}Rule} \notin \{\mathbf{left\text{-}either}, \mathbf{right\text{-}either}\}$$

in 4.8. We say that the propositions in $FA(D)$ are *strictly used* in D.

Note that the computation of $FA(D)$ is not trivial, meaning that it cannot proceed in a local manner down the abstract syntax tree of D using only a fixed amount of memory: a variable amount of state must be maintained in order to deal with clauses 4.9 and 4.10. The latter clause, in particular, is especially computation-intensive because it also calls for the computation of $\mathcal{C}(D_1)$. The reader should reflect on what would be involved in computing, for instance, $FA(D)$ for a D of the form $D_1; \cdots; D_{100}$. In fact we will see shortly that evaluating D in a given β can be reduced to the computation of $FA(D)$.

Theorem 4.10 *For any well-formed D, $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ iff $\beta \supseteq FA(D)$.*

Proof: We first prove that $\beta \supseteq FA(D)$ whenever $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ by strong induction on the length n of the derivation of the latter judgment. When $n = 1$ the judgment must be an instance of $[R_1]$, or $[R_2]$, or $[R_3]$, or one of the primitive deduction axioms listed in Figure 4.3, and $\beta \supseteq FA(D)$ can be readily verified. When $n > 1$ we distinguish the same cases as in the proof of Theorem 4.8:

$[R_4]$: In that case D is of the form **assume** P **in** D' where $\mathcal{C}(D) = P \Rightarrow Q$ and the judgment $\beta \cup \{P\} \vdash D' \rightsquigarrow Q$ is derivable in fewer than n steps. Now $FA(D) = FA(D') - \{P\}$, so we need to show

$$FA(D') - \{P\} \subseteq \beta. \tag{4.11}$$

From the inductive hypothesis,

$$\beta \cup \{P\} \supseteq FA(D'). \tag{4.12}$$

Pick any $R \in FA(D') - \{P\}$, so that $R \in FA(D')$, $R \neq P$. From 4.12,

$$R \in \beta \cup \{P\}$$

i.e., either $R \in \beta$ or $R = P$. Since the latter is impossible, we must have $R \in \beta$. We have thus shown that $R \in \beta$ whenever $R \in FA(D') - \{P\}$, which proves 4.11.

[R_5]: In that case $D = D_1; D_2$ where $\mathcal{C}(D) = P_2$ and the judgments $\beta \vdash D_1 \rightsquigarrow P_1$ and $\beta \cup \{P_1\} \vdash D_2 \rightsquigarrow P_2$ are derivable in fewer than n steps. Here $FA(D) = FA(D_1) \cup [FA(D_2) - \{P_1\}]$ (since $P_1 = \mathcal{C}(D_1)$), so we need to show

$$FA(D_1) \cup [FA(D_2) - \{P_1\}] \subseteq \beta. \tag{4.13}$$

By the inductive hypothesis we have

$$FA(D_1) \subseteq \beta, FA(D_2) \subseteq \beta \cup \{P_1\}$$

and from these 4.13 follows directly.

The converse direction—that $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ whenever $FA(D) \subseteq \beta$—can be proved by induction on the structure of D. ■

Accordingly:

Corollary 4.11 *For well-formed D, $Eval(D, \beta) = \mathcal{C}(D)$ iff $FA(D) \subseteq \beta$. Equivalently, $Eval(D, \beta) = error$ iff there is some $P \in FA(D)$ such that $P \notin \beta$.*

The above corollary captures the sense in which evaluation can be reduced to the computation of *FA*: to compute $Eval(D, \beta)$, for any given D and β, simply compute $FA(D)$ and $\mathcal{C}(D)$: if $FA(D) \subseteq \beta$, output $\mathcal{C}(D)$, otherwise output *error*. Intuitively, this reduction is the reason why the computation of $FA(D)$ cannot be much easier than the evaluation of D in a given assumption base.

It is now straightforward to prove Lemma 4.13 below, which is a useful technical tool and which formalizes the intuition that, in evaluating a deduction D in an assumption base β, it is only those propositions which are strictly used in D whose presence in β matters; the presence—or absence—of other propositions is irrelevant. More precisely, let us say that two assumption bases β_1 and β_2 *agree* on a set of propositions Φ, written $\beta_1 \equiv_\Phi \beta_2$, whenever

$$\forall P \in \Phi,\ P \in \beta_1 \text{ iff } P \in \beta_2$$

or equivalently, iff $\beta_1 \cap \Phi = \beta_2 \cap \Phi$. The following lemma captures the essential properties of this relation:

Lemma 4.12 $\equiv_\Phi$ *is an equivalence relation which partitions the set of all assumption bases into* $2^{|\Phi|}$ *equivalence classes. Each of those contains a unique subset of* Φ*, which may be taken as a representative of the equivalence class. Furthermore:* (a) if $\beta_1 \equiv_\Phi \beta_2$ then $\beta_1 \equiv_\Psi \beta_2$ for all $\Psi \subseteq \Phi$; (b) if $\beta_1 \equiv_\Phi \beta_2$ then $\beta_1 \cup \beta \equiv_\Phi \beta_2 \cup \beta$; (c) if $\beta_1 \equiv_{\Phi - \Psi} \beta_2$ then $\beta_1 \cup \Psi \equiv_{\Phi \cup \Psi} \beta_2 \cup \Psi$.

Lemma 4.13 (Strictness Coincidence Lemma) *If* $\beta_1 \equiv_{FA(D)} \beta_2$ *then*

$$(\forall P)\ [\beta_1 \vdash D \rightsquigarrow P \quad \textit{iff} \quad \beta_2 \vdash D \rightsquigarrow P].$$

Equivalently, if $\beta_1 \equiv_{FA(D)} \beta_2$ *then* $\mathit{Eval}(D, \beta_1) = \mathit{Eval}(D, \beta_2)$.

Proof: By virtue of $\beta_1 \equiv_{FA(D)} \beta_2$ it follows that $\beta_1 \supseteq FA(D)$ iff $\beta_2 \supseteq FA(D)$. Therefore, from Corollary 4.11, if $\beta_1 \supseteq FA(D)$ then

$$\mathit{Eval}(D, \beta_1) = \mathcal{C}(D) = \mathit{Eval}(D, \beta_2)$$

otherwise $\mathit{Eval}(D, \beta_1) = \mathit{error} = \mathit{Eval}(D, \beta_2)$. ■

4.4 Examples

In this section we present a variety of sample $\mathcal{CND}$ deductions. The first set of examples consists of deductions which derive their conclusions from no premises whatsoever. Accordingly, these deductions can be successfully evaluated in the empty assumption base.[3] Their results are called "tautologies", or "logically valid", because it is impossible for them to be false. This comes about as a result of the soundness of $\mathcal{CND}$, which we will prove in Section 4.6: if a deduction D produces P in an assumption base β—i.e., if $\beta \vdash D \rightsquigarrow P$—then P is a logical consequence of β, written $\beta \models P$. This means that it is impossible for the members of β to be true without P also being true. But if β is the empty set then there are no members to speak of, and the preceding statement entails that it is impossible for P not to be true.

The reader is invited to read a few of these—contrasting them with their analogues in other systems—and tackle the rest on his own.

[3] Such deductions are sometimes called "categorical" [61]; but it is more customary to reserve the term "proof" for them. I.e., a proof of a proposition P is a deduction that derives P from the empty set of premises. And the term "theorem" is reserved precisely for those propositions—those that are deducible from the empty set of premises. In this document we use the terms "proof" and "deduction" interchangeably; and we usually speak of a theorem with respect to a given set of propositions, e.g., "P is a theorem of β", meaning that P is deducible from β.

$$\boxed{P \Rightarrow \neg\neg P}$$

Proof:
assume P **in**
  suppose-absurd $\neg P$ **in**
    absurd $P, \neg P$

$$\boxed{(P \Rightarrow Q) \Rightarrow [(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)]}$$

Proof:
assume $P \Rightarrow Q$ **in**
  assume $Q \Rightarrow R$ **in**
    assume P **in**
      begin
        modus-ponens $P \Rightarrow Q, P$;
        modus-ponens $Q \Rightarrow R, Q$
      end

$$\boxed{[P \Rightarrow (Q \Rightarrow R)] \Rightarrow [Q \Rightarrow (P \Rightarrow R)]}$$

Proof:
assume $P \Rightarrow (Q \Rightarrow R)$ **in**
  assume Q **in**
    assume P **in**
      begin
        modus-ponens $P \Rightarrow (Q \Rightarrow R), P$;
        modus-ponens $Q \Rightarrow R, Q$
      end

$$\boxed{[P \Rightarrow (Q \Rightarrow R)] \Rightarrow [(P \wedge Q) \Rightarrow R]}$$

Proof:
assume $P \Rightarrow (Q \Rightarrow R)$ **in**
  assume $P \wedge Q$ **in**
    begin

    left-and $P \wedge Q$;
    modus-ponens $P \Rightarrow (Q \Rightarrow R), P$;
    right-and $P \wedge Q$;
    modus-ponens $Q \Rightarrow R, Q$
  end

$$\boxed{[(P \wedge Q) \Rightarrow R] \Rightarrow [P \Rightarrow (Q \Rightarrow R)]}$$

Proof:

assume $(P \wedge Q) \Rightarrow R$ **in**
  assume P **in**
    assume Q **in**
      begin
        both P, Q;
        modus-ponens $(P \wedge Q) \Rightarrow R, P \wedge Q$
      end

$$\boxed{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)}$$

Proof:

assume $P \Rightarrow Q$ **in**
  assume $\neg Q$ **in**
    modus-tollens $P \Rightarrow Q, \neg Q$;

$$\boxed{(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)}$$

Proof:

assume $\neg Q \Rightarrow \neg P$ **in**
  assume P **in**
    begin
      suppose-absurd $\neg Q$ **in**
        begin
          modus-ponens $\neg Q \Rightarrow \neg P, \neg Q$;
          absurd $P, \neg P$
        end;

```
      double-negation ¬¬Q
    end
```

$$\boxed{(P \lor Q) \Rightarrow (Q \lor P)}$$

Proof:

```
assume P ∨ Q in
  begin
    assume P in
      right-either Q, P;
    assume Q in
      left-either Q, P;
    constructive-dilemma P ∨ Q, P ⇒ Q ∨ P, Q ⇒ Q ∨ P
  end
```

$$\boxed{(\neg P \lor Q) \Rightarrow (P \Rightarrow Q)}$$

Proof:

```
assume ¬P ∨ Q in
  assume P in
    begin
      suppose-absurd ¬Q in
        begin
          assume ¬P in
            absurd P, ¬P;
          assume Q in
            absurd Q, ¬Q;
          constructive-dilemma ¬P ∨ Q, ¬P ⇒ false, Q ⇒ false
        end;
      double-negation ¬¬Q
    end
```

$$\boxed{\neg(P \lor Q) \Rightarrow (\neg P \land \neg Q)}$$

Proof:

assume $\neg(P \vee Q)$ **in**
  begin
    suppose-absurd P **in**
      begin
        left-either P, Q;
        absurd $P \vee Q, \neg(P \vee Q)$
      end;
    suppose-absurd Q **in**
      begin
        right-either P, Q;
        absurd $P \vee Q, \neg(P \vee Q)$
      end;
    both $\neg P, \neg Q$
  end

$$\boxed{(\neg P \vee \neg Q) \Rightarrow \neg(P \wedge Q)}$$

Proof:

assume $\neg P \vee \neg Q$ **in**
  suppose-absurd $P \wedge Q$ **in**
    begin
      left-and $P \wedge Q$;
      right-and $P \wedge Q$;
      assume $\neg P$ **in**
        absurd $P, \neg P$;
      assume $\neg Q$ **in**
        absurd $Q, \neg Q$;
      constructive-dilemma $\neg P \vee \neg Q, \neg P \Rightarrow \textbf{false}, \neg Q \Rightarrow \textbf{false}$
    end

$$\boxed{(\neg P \wedge \neg Q) \Rightarrow \neg(P \vee Q)}$$

Proof:

assume $\neg P \wedge \neg Q$ **in**
  begin

left-and $\neg P \wedge \neg Q$;
right-and $\neg P \wedge \neg Q$;
assume P **in**
  absurd $P, \neg P$;
assume Q **in**
  absurd $Q, \neg Q$;
suppose-absurd $P \vee Q$ **in**
  constructive-dilemma $P \vee Q, P \Rightarrow \textbf{false}, Q \Rightarrow \textbf{false}$
end

$$\boxed{P \Leftrightarrow [P \wedge (P \vee Q)]}$$

Proof:
assume P **in**
  begin
    left-either P, Q;
    both $P, P \vee Q$
  end;
assume $P \wedge (P \vee Q)$ **in**
  left-and $P \wedge (P \vee Q)$;
equivalence $P \Rightarrow [P \wedge (P \vee Q)], [P \wedge (P \vee Q)] \Rightarrow P$

$$\boxed{[(P \wedge Q) \vee (\neg P \wedge \neg Q)] \Rightarrow (P \Leftrightarrow Q)}$$

Proof:
assume $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ **in**
  begin
    assume $P \wedge Q$ **in**
      begin
        left-and $P \wedge Q$;
        right-and $P \wedge Q$;
        assume P **in** Q;
        assume Q **in** P;
        equivalence $P \Rightarrow Q, Q \Rightarrow P$
      end;
    assume $\neg P \wedge \neg Q$ **in**

begin
  left-and $\neg P \wedge \neg Q$;
  right-and $\neg P \wedge \neg Q$;
  assume P **in**
    begin
      suppose-absurd $\neg Q$ **in**
        absurd $P, \neg P$;
      double-negation $\neg\neg Q$
    end;
  assume Q **in**
    begin
      suppose-absurd $\neg P$ **in**
        absurd $Q, \neg Q$;
      double-negation $\neg\neg P$
    end;
  equivalence $P \Rightarrow Q, Q \Rightarrow P$
end;
constructive-dilemma $(P \wedge Q) \vee (\neg P \wedge \neg Q)$,$(P \wedge Q) \Rightarrow (P \Leftrightarrow Q)$,
$(\neg P \wedge \neg Q) \Rightarrow (P \Leftrightarrow Q)$;
end

The second set of examples comprises deductions that derive a certain conclusion P from a set of given premises $P_1, \ldots, P_n$. Such a deduction can be successfully evaluated in any assumption base that contains the premises. Here we use the primitive **assert**, which is not part of $\mathcal{CND}$ proper but is rather a “top-level directive” that would be useful to have in an implementation (similar, say, to **define** in a Scheme environment). The effect of **assert** is to insert a given proposition into the current assumption base. Therefore, to verify the claim that D produces a conclusion P from premises $P_1, \ldots, P_n$, we use **assert** to add the premises $P_1, \ldots, P_n$ into the current assumption base and proceed to evaluate D.

Premise 1: $P \vee (Q \wedge R)$
Premise 2: $(P \Rightarrow S) \wedge (S \Rightarrow R)$

Conclusion: R

Proof:
  assert $P \vee (Q \wedge R)$;

assert $(P \Rightarrow S) \wedge (S \Rightarrow R)$;
left-and $(P \Rightarrow S) \wedge (S \Rightarrow R)$;
right-and $(P \Rightarrow S) \wedge (S \Rightarrow R)$;
assume P **in**
  begin
    modus-ponens $P \Rightarrow S, P$;
    modus-ponens $S \Rightarrow R, S$
  end;
assume $Q \wedge R$ **in**
  right-and $Q \wedge R$;
constructive-dilemma $P \vee (Q \wedge R), P \Rightarrow R, (Q \wedge R) \Rightarrow R$

Premise 1: $\neg P \Rightarrow Q$
Premise 2: $Q \Rightarrow P$

Conclusion: P

Proof:
assert $\neg P \Rightarrow Q$;
assert $Q \Rightarrow P$;
suppose-absurd $\neg P$ **in**
  begin
    modus-ponens $\neg P \Rightarrow Q, \neg P$;
    modus-ponens $Q \Rightarrow P, Q$;
    absurd $P, \neg P$
  end;
double-negation $\neg\neg P$

Premise 1: $(P \wedge \neg Q) \Rightarrow R$
Premise 2: $R \Rightarrow Q$

Conclusion: $P \Rightarrow Q$

Proof:
assert $(P \wedge \neg Q) \Rightarrow R$;
assert $R \Rightarrow Q$;
assume P **in**
  begin

suppose-absurd $\neg Q$ **in**
begin
both $P, \neg Q$;
modus-ponens $(P \wedge \neg Q) \Rightarrow R, P \wedge \neg Q$;
modus-ponens $R \Rightarrow Q, R$;
absurd $Q, \neg Q$
end;
double-negation $\neg\neg Q$
end

$$\frac{\text{Premise 1: } [P \wedge (Q \vee R)] \Rightarrow (Q \wedge R)}{\text{Conclusion: } P \Rightarrow (Q \Rightarrow R)}$$

Proof:
assert $[P \wedge (Q \vee R)] \Rightarrow (Q \wedge R)$;
assume P **in**
assume Q **in**
begin
left-either Q, R;
both $P, Q \vee R$;
modus-ponens $[P \wedge (Q \vee R)] \Rightarrow (Q \wedge R), P \wedge (Q \vee R)$;
right-and $Q \wedge R$
end

$$\frac{\begin{array}{l}\text{Premise 1: } (P_1 \vee P_2) \Rightarrow (P_3 \wedge P_4) \\ \text{Premise 2: } (P_4 \vee P_5) \Rightarrow P_6\end{array}}{\text{Conclusion: } P_1 \Rightarrow P_6}$$

Proof:
assert $(P_1 \vee P_2) \Rightarrow (P_3 \wedge P_4)$;
assert $(P_4 \vee P_5) \Rightarrow P_6$;
assume P_1 **in**
begin
left-either P_1, P_2;
modus-ponens $(P_1 \vee P_2) \Rightarrow (P_3 \wedge P_4), P_1 \vee P_2$;
right-and $P_3 \wedge P_4$;

left-either P_4, P_5;
modus-ponens $(P_4 \vee P_5) \Rightarrow P_6, P_4 \vee P_5$
end

The next lemma will come handy in the completeness proof:

Lemma 4.14 *We have:*

(a) $\Phi \vdash_{\mathcal{CND}} P$ *iff* $\Phi \vdash_{\mathcal{CND}} \neg\neg P$*;*

(b) $\Phi \vdash_{\mathcal{CND}} P \wedge Q$ *iff* $\Phi \vdash_{\mathcal{CND}} P$ *and* $\Phi \vdash_{\mathcal{CND}} Q$*;*

(c) *if* $\Phi \vdash_{\mathcal{CND}} P$ *or* $\Phi \vdash_{\mathcal{CND}} Q$ *then* $\Phi \vdash_{\mathcal{CND}} P \vee Q$*;*

(d) *if* $\Phi \vdash_{\mathcal{CND}} P \Rightarrow Q$ *and* $\Phi \vdash_{\mathcal{CND}} P$ *then* $\Phi \vdash_{\mathcal{CND}} Q$*;*

(e) $\Phi \vdash_{\mathcal{CND}} P \Leftrightarrow Q$ *iff* $\Phi \vdash_{\mathcal{CND}} P \Rightarrow Q$ *and* $\Phi \vdash_{\mathcal{CND}} Q \Rightarrow P$*;*

(f) *if* $\Phi \vdash_{\mathcal{CND}} \neg P$ *and* $\Phi \vdash_{\mathcal{CND}} \neg Q$ *then* $\Phi \vdash_{\mathcal{CND}} \neg(P \vee Q)$*.*

Proof: For (a), suppose $\Phi \vdash_{\mathcal{CND}} P$, so that $\beta \vdash D \rightsquigarrow P$ for some $\beta \subseteq \Phi$. Letting

$$D' = D; \textbf{suppose-absurd } \neg P \textbf{ in absurd } P, \neg P$$

we get

$$\beta \vdash D' \rightsquigarrow \neg\neg P$$

and thus $\Phi \vdash_{\mathcal{CND}} \neg\neg P$. For the converse, if $\Phi \vdash_{\mathcal{CND}} \neg\neg P$ then $\beta \vdash D \rightsquigarrow \neg\neg P$ for some D and $\beta \subseteq \Phi$, so

$$\beta \vdash D; \textbf{double-negation } \neg\neg P \rightsquigarrow P$$

and thus $\Phi \vdash_{\mathcal{CND}} P$. Similar reasoning using the primitive rules **both**, **left-and**, **right-and**, **left-either**, **right-either**, **mp**, **equivalence**, **left-iff**, and **right-iff**, will establish (b), (c), (d), and (e). For (f), suppose that $\Phi \vdash_{\mathcal{CND}} \neg P$ and $\Phi \vdash_{\mathcal{CND}} \neg Q$. From (b), we get $\Phi \vdash_{\mathcal{CND}} \neg P \wedge \neg Q$, so that

$$\beta \vdash D \rightsquigarrow \neg P \wedge \neg Q$$

for some D and $\beta \subseteq \Phi$. Now let D' be the deduction of $(\neg P \wedge \neg Q) \Rightarrow \neg(P \vee Q)$ given above. Set

$$D'' = D'; D; \textbf{modus-ponens } (\neg P \wedge \neg Q) \Rightarrow \neg(P \vee Q), \neg P \wedge \neg Q.$$

Now

$$\beta \vdash D'' \rightsquigarrow \neg(P \vee Q)$$

hence $\Phi \vdash_{\mathcal{CND}} \neg(P \vee Q)$. ■

4.5 Proof equivalence

Let us say that two deductions D_1 and D_2 are *observationally equivalent with respect to an assumption base* β, written $D_1 \approx_\beta D_2$, whenever

$$\beta \vdash D_1 \rightsquigarrow P \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow P \tag{4.14}$$

for all P. For instance,

$$D_1 = \textbf{left-either } A, B \quad \text{and} \quad D_2 = \textbf{right-either } A, B$$

are observationally equivalent with respect to any assumption base that either contains both A and B or neither of them. We clearly have:

Lemma 4.15 *$D_1 \approx_\beta D_2$ iff $Eval(D_1, \beta) = Eval(D_2, \beta)$.*

Thus D_1 and D_2 are observationally equivalent in a given β iff when we evaluate them in β we obtain the same result: either the same conclusion, or an error in both cases.

If we have $D_1 \approx_\beta D_2$ for *all* β then we write $D_1 \approx D_2$ and say that D_1 and D_2 are *observationally equivalent*. For a simple example, we have

$$\textbf{left-iff } A \Leftrightarrow A \;\approx\; \textbf{right-iff } A \Leftrightarrow A$$

as well as

$$\begin{array}{lcl} \textbf{left-and } A \wedge B; & & \textbf{right-and } A \wedge B; \\ \textbf{right-and } A \wedge B; & \approx & \textbf{left-and } A \wedge B; \\ \textbf{both } B, A & & \textbf{both } B, A \end{array}$$

The reader will verify that $\approx$ is an equivalence relation.

With this framework in place, we can now precisely formulate and prove statements such as our earlier assertion that "composition is not associative":

Theorem 4.16 $D_1; (D_2; D_3) \not\approx (D_1; D_2); D_3$.

Proof: Take $D_1 = \textbf{double-negation } \neg\neg A$, $D_2 = \textbf{true}$, $D_3 = A$, and consider any β that contains $\neg\neg A$ but not A. ■

We note, however, that composition is idempotent (i.e., $D \approx D; D$) and that associativity does hold in the case of claims: $P_1; (P_2; P_3) \approx (P_1; P_2); P_3$. Commutativity fails in all cases. A certain kind of distributivity holds between the constructor **assume** and the composition operator (as well as between **suppose-absurd** and composition), in the following sense:

Lemma 4.17 **assume** P **in** $(D_1; D_2) \approx D_1;$ **assume** P **in** D_2 *whenever* $P \notin FA(D_1)$.

This lemma will form the basis for our "hoisting" transformation in the next chapter. Further, we note that $\approx$ is compatible with the abstract syntax constructors of $\mathcal{CND}$:

Lemma 4.18 (Compatibility) *If* $D \approx D'$ *then*

$$\textbf{assume } P \textbf{ in } D \approx \textbf{assume } P \textbf{ in } D'.$$

In addition, if $D_1 \approx D_1'$ *and* $D_2 \approx D_2'$ *then* $D_1; D_2 \approx D_1'; D_2'$.

From Lemma 4.15 it is clear that the relation $\approx_\beta$ is decidable: to determine whether $D_1 \approx_\beta D_2$, we evaluate each deduction in β and compare the results for equality. But in fact even the more general relation $\approx$ is decidable, because even though $\approx$ is defined by quantification over *all* assumption bases, the Strictness Coincidence Lemma (Lemma 4.13) entails that only the finitely many propositions which are strictly used in D_1 and D_2 are relevant in the decision. Theorem 4.20 below indirectly furnishes an algorithm for deciding observational equivalence.

For well-formed deductions, a necessary—but not sufficient—condition for observational equivalence is conclusion identity:

Lemma 4.19 $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ *whenever* $D_1 \approx D_2$.

Proof: Set $\beta = FA(D_1) \cup FA(D_2)$, so that $\beta \supseteq FA(D_1)$, $\beta \supseteq FA(D_2)$. By Corollary 4.11,

$$Eval(D_1, \beta) = \mathcal{C}(D_1) \tag{4.15}$$

and

$$Eval(D_2, \beta) = \mathcal{C}(D_2). \tag{4.16}$$

But $Eval(D_1, \beta) = Eval(D_2, \beta)$ by the supposition $D_1 \approx D_2$, hence $\mathcal{C}(D_1) = \mathcal{C}(D_2)$. ■

Note that this does not hold for observational equivalence in the context of a particular assumption base β: we may have $D_1 \approx_\beta D_2$ even if $\mathcal{C}(D_1) \neq \mathcal{C}(D_2)$ (this will be the case if evaluating both deductions in β results in *error*). Also, as the counter-example in the proof of Theorem 4.16 showed, $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ does not suffice for $D_1 \approx D_2$. The problem lies in differences of strict (free) assumptions; the following result shows that $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ *in conjunction* with $FA(D_1) = FA(D_2)$ *is* sufficient for observational equivalence:

Theorem 4.20 $D_1 \approx D_2$ *iff* $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ *and* $FA(D_1) = FA(D_2)$. *Accordingly, observational equivalence is decidable.*

Proof: First we show that if $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $FA(D_1) = FA(D_2)$ then $D_1 \approx D_2$. Pick any β: either $\beta \supseteq FA(D_1) = FA(D_2)$ or not. If so, then by Corollary 4.11,

$$Eval(D_1, \beta) = \mathcal{C}(D_1) \text{ and } Eval(D_2, \beta) = \mathcal{C}(D_2)$$

hence

$$Eval(D_1, \beta) = Eval(D_2, \beta)$$

by the supposition $\mathcal{C}(D_1) = \mathcal{C}(D_2)$. On the other hand, if $\beta \not\supseteq FA(D_1) = FA(D_2)$ then

$$Eval(D_1, \beta) = error = Eval(D_2, \beta).$$

Therefore, $Eval(D_1, \beta) = Eval(D_2, \beta)$ for all β, which is to say $D_1 \approx D_2$.

Conversely, suppose $D_1 \approx D_2$. Then $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ by Lemma 4.19. Moreover, by supposition,

$$Eval(D_1, FA(D_1)) = Eval(D_2, FA(D_1)) \tag{4.17}$$

and

$$Eval(D_1, FA(D_2)) = Eval(D_2, FA(D_2)). \tag{4.18}$$

By Corollary 4.11, $Eval(D_1, FA(D_1)) = \mathcal{C}(D_1)$, hence from 4.17,

$$Eval(D_2, FA(D_1)) \neq error.$$

Therefore, by Corollary 4.11,

$$FA(D_2) \subseteq FA(D_1). \tag{4.19}$$

Likewise, $Eval(D_2, FA(D_2)) = \mathcal{C}(D_2)$, hence, from 4.18, $Eval(D_1, FA(D_2)) \neq error$, and thus we must have

$$FA(D_1) \subseteq FA(D_2). \tag{4.20}$$

From 4.19 and 4.20 we get $FA(D_1) = FA(D_2)$. ■

Relaxing observational equivalence

Observational equivalence is a very strong condition. Oftentimes we are only interested in replacing a deduction D_1 by some D_2 on the assumption that D_1 will yield its conclusion in the intended β (i.e., on the assumption that its evaluation will not lead to error), even though we might have $D_1 \not\approx D_2$. To take a simple example, although we have $P; D \not\approx D$ (pick D to be the claim **true** and consider any β that does not contain P), it is true that in any given assumption base, *if* $P; D$ produces a conclusion Q *then* so will D. (In fact this observation will be the formal justification behind an "optimization" we will introduce later for removing redundant claims.) We formalize this relation as follows.

We write $D_1 \rightarrowtail_\beta D_2$ to mean that, for all P,

$$\text{if } \beta \vdash D_1 \rightsquigarrow P \text{ then } \beta \vdash D_2 \rightsquigarrow P.$$

That is, $D_1 \rightarrowtail_\beta D_2$ holds if $\mathit{Eval}(D_1, \beta) = \mathit{Eval}(D_2, \beta)$ whenever $\mathit{Eval}(D_1, \beta) = \mathcal{C}(D_1)$, or, equivalently, whenever $\mathit{Eval}(D_1, \beta) \neq \mathit{error}$. And we will write $D_1 \rightarrowtail D_2$ to mean that $D_1 \rightarrowtail_\beta D_2$ for *all* β.

Clearly, $\rightarrowtail$ is not a symmetric relation: we vacuously have

$$\textbf{suppose-absurd } A \textbf{ in } A \rightarrowtail \textbf{true}$$

but the converse does not hold. However, $\rightarrowtail$ is a quasi-order (reflexive and transitive), and in fact, as the reader will verify, $\approx$ is the contensive equality generated by the weaker relation's symmetric closure. (Observe that this also follows from Theorem 4.20 and Theorem 4.23; alternatively, Theorem 4.20 could have been established by independent proofs of Lemma 4.21 and Theorem 4.23).

Lemma 4.21 *$\rightarrowtail$ is a quasi-order whose symmetric closure coincides with $\approx$. Accordingly, $D_1 \approx D_2$ iff $D_1 \rightarrowtail D_2$ and $D_2 \rightarrowtail D_1$.*

It will be useful to note that $\rightarrowtail$ is compatible with the syntactic constructs of $\mathcal{CND}$:

Lemma 4.22 *If $D_1 \rightarrowtail D_1'$ and $D_2 \rightarrowtail D_2'$ then*

(a) **assume** P **in** $D_1 \rightarrowtail$ **assume** P **in** D_1';

(b) $D_1; D_2 \rightarrowtail D_1'; D_2'$.

Proof: Pick any assumption base β and proposition Q. For (a), supposing that

$$\beta \vdash \textbf{assume } P \textbf{ in } D_1 \rightsquigarrow Q,$$

Q must be of the form $P \Rightarrow P'$, where $\beta \cup \{P\} \vdash D_1 \rightsquigarrow P'$. Since $D_1 \rightarrowtail D_1'$, we have

$$\beta \cup \{P\} \vdash D_1' \rightsquigarrow P'$$

hence, by the semantics of $\mathcal{CND}$,

$$\beta \vdash \textbf{assume } P \textbf{ in } D_1' \rightsquigarrow (P \Rightarrow P') = Q.$$

For (b), if $\beta \vdash D_1; D_2 \leadsto Q$ then there is a P_1 such that

$$\beta \vdash D_1 \leadsto P_1 \tag{4.21}$$

and

$$\beta \cup \{P_1\} \vdash D_2 \leadsto Q \tag{4.22}$$

Since $D_1 \rightarrowtail D_1'$ and $D_2 \rightarrowtail D_2'$, 4.21 and 4.22 imply, respectively, that $\beta \vdash D_1' \leadsto P_1$ and

$$\beta \cup \{P_1\} \vdash D_2' \leadsto Q.$$

Hence, by the semantics of composition we get $\beta \vdash D_1'; D_2' \leadsto Q$. ■

Reasoning similar to that used in the proof of Theorem 4.20 will show:

Theorem 4.23 *$D_1 \rightarrowtail D_2$ iff $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $FA(D_1) \supseteq FA(D_2)$. Therefore, the relation $\rightarrowtail$ is decidable.*

Finally, let us define a last equivalence relation $\equiv$ on (well-formed) deductions as $D_1 \equiv D_2$ iff $\mathcal{C}(D_1) = \mathcal{C}(D_2)$. This is the weakest of all three relations, and the easiest to compute. It is instructive to contrast the relations $\approx$, $\rightarrowtail$, and $\equiv$ in the following light: Suppose we are "optimizing" a collection of deductions in the same spirit that a compiler might optimize a source program, and we are considering the replacement of a certain D_1 by another (hopefully more efficient) deduction D_2. Which of the three relations should we wish to obtain between D_1 and its replacement D_2? Questions of this sort have practical import for implemented DPLs.

For the replacement to be perfectly safe we should insist on observational equivalence. For $D_1 \approx D_2$ means that the two deductions behave identically in all contexts, i.e., in all assumption bases. For any β, if D_1 fails in β then D_2 will fail in β as well; while if D_1 produces a conclusion P in β, then D_2 will produce that same conclusion in β. Hence the replacement preserves the semantics completely.

If we only have $D_1 \rightarrowtail D_2$ then the replacement will be safe provided that D_1 does not fail in the intended assumption base. This, of course, depends on what the intended assumption base is. For instance, suppose $D_1 = A;$ **right-and** $A \wedge B$ and $D_2 =$ **right-and** $A \wedge B$, so that $D_1 \rightarrowtail D_2$. Now if the specification of D_1 stipulates that we may only take $A \wedge B$ as a given premise, i.e., that $A \wedge B$ is the only proposition whose presence in the assumption base we can take for granted, then replacing D_1 by D_2 does not completely preserve the semantics, because, for instance, $Eval(D_1, \{A \wedge B\}) = error$ whereas $Eval(D_2, \{A \wedge B\}) = B$. However, the replacement is certainly conservative in the sense that D_2 preserves the conclusion of D_1 *and* does not introduce any additional free assumptions. That is always the case by virtue of Theorem 4.23: if $D_1 \rightarrowtail D_2$ then $D_1 \equiv D_2$ (conclusion is preserved), and

$FA(D_1) \supseteq FA(D_2)$ (no additional free assumptions are introduced). In that sense, replacing D_1 by D_2 is a safe transformation, and can be confidently carried out whenever we are assured of the correctness of D_1, or conditionally carried out on that explicit assumption.

Finally, if we only have $D_1 \equiv D_2$ then the replacement is not safe in any context, even if we take the correctness of D_1 for granted. The reason is that D_2 might introduce free assumptions which do not occur in D_1. For instance, for D_1 and D_2 as in the last paragraph, we have $D_2 \equiv D_1$, yet replacing D_2 by D_1 is clearly unsafe.

We summarize the results of this section as follows:

Theorem 4.24 *We have $\approx\ \supseteq\ \rightarrowtail\ \supseteq\ \equiv$. That is, $D_1 \approx D_2$ implies $D_1 \rightarrowtail D_2$, and $D_1 \rightarrowtail D_2$ implies $D_1 \equiv D_2$. Moreover, these inclusions are proper: $D_1 \rightarrowtail D_2$ does not imply $D_1 \approx D_2$, and $D_1 \equiv D_2$ implies neither $D_1 \approx D_2$ nor $D_1 \rightarrowtail D_2$. However, $D_1 \equiv D_2$ in tandem with $FA(D_1) \supseteq FA(D_2)$ implies $D_1 \rightarrowtail D_2$, and $D_1 \equiv D_2$ in tandem with $FA(D_1) = FA(D_2)$ implies $D_1 \approx D_2$.*

4.6 Metatheory

We will view the set of all propositions as the free term algebra formed by the constructors $\neg$, $\wedge$, etc., over the set of atoms (treating the latter as variables). Any Boolean algebra $\mathcal{B}$, say the one with carrier $\{0,1\}$, can be seen as a $\{\mathbf{true}, \mathbf{false}, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$-algebra with respect to the expected realizations ($\wedge^{\mathcal{B}}(x,y) = \min\{x,y\}$, $\vee^{\mathcal{B}}(x,y) = \max\{x,y\}$, $\mathbf{false}^{\mathcal{B}} = 0$, etc.). Now by an *interpretation* $\mathcal{I}$ we will mean a function from the set of atoms to $\{0,1\}$. We will write $\widehat{\mathcal{I}}$ for the unique homomorphic extension of $\mathcal{I}$ to the set of all propositions. Since $\widehat{\mathcal{I}}$ is completely determined by $\mathcal{I}$, we may, for most purposes, conflate the two.

If $\mathcal{I}(P) = 1$ ($\mathcal{I}(P) = 0$) we say that $\mathcal{I}$ *satisfies* (*falsifies*) P, or that it is a *model* of it. This is written as $\mathcal{I} \models P$. We call P *satisfiable* if it satisfied by some interpretation; a *tautology* (or *valid*) if it is satisfied by all interpretations; and *unsatisfiable* (or a *contradiction*) if it is not satisfied by any interpration. If $\mathcal{I} \models P$ for every $P \in \Phi$ then we write $\mathcal{I} \models \Phi$ and say that $\mathcal{I}$ satisfies (or is a model of) Φ. We regard the empty set of propositions as (vacuously) satisfied by every interpretation. We will call Φ satisfiable or unsatisfiable according to whether or not it has a model. We write $\Phi_1 \models \Phi_2$ to indicate that every model of Φ_1 is also a model of Φ_2. If this is the case we say that the elements of Φ_2 are *logical consequences* of (or are "logically implied by") the propositions in Φ_1. A single proposition P may appear in place of either Φ_1 or Φ_2 as an abbreviation for the singleton $\{P\}$.

The following lemma is a direct consequence of the above definitions:

Lemma 4.25 *If $\Phi \cup \{P_1\} \models P_2$ then $\Phi \models P_1 \Rightarrow P_2$.*

Theorem 4.26 *If* $\beta \vdash D \rightsquigarrow P$ *then* $\beta \models P$.

Proof: By induction on the structure of D. When D is a claim or a primitive deduction, the supposition that the judgment $\beta \vdash D \rightsquigarrow P$ is derivable means that the judgment in question is either an instance of $[R_1]$, or $[R_2]$, or $[R_3]$; or an instance of one of the primitive-deduction axioms of Figure 4.3. In any of these cases it is readily verified by inspection that $\beta \models P$.

When D is a hypothetical deduction of the form **assume** P_1 **in** D', the supposition $\beta \vdash D \rightsquigarrow P$ means that $P = P_1 \Rightarrow P_2$ and $\beta \cup \{P_1\} \vdash D' \rightsquigarrow P_2$. By the inductive hypothesis, $\beta \cup \{P_1\} \models P_2$. Therefore, by Lemma 4.25, $\beta \models P_1 \Rightarrow P_2$, i.e., $\beta \models P$. Finally, when D is a composite deduction $D_1; D_2$, we must have $\beta \vdash D_1 \rightsquigarrow P_1$ and

$$\beta \cup \{P_1\} \vdash D_2 \rightsquigarrow P_2$$

where $P = P_2$. Inductively,

$$\beta \models P_1 \tag{4.23}$$

and

$$\beta \cup \{P_1\} \models P_2 = P \tag{4.24}$$

so the desired $\beta \models P$ now follows from 4.23 and 4.24 by the transitivity of $\models$. ■

We immediately get:

Corollary 4.27 (Soundness) *If* $\Phi \vdash_{\mathcal{CND}} P$ *then* $\Phi \models P$.

Next we consider completeness. A set of propositions Φ will be called *saturated* iff for all P and Q:

- $P \in \Phi$ iff $\neg P \notin \Phi$. That is, exactly one of the couple $\{P, \neg P\}$ is in Φ.
- $P \wedge Q \in \Phi$ iff both $P \in \Phi$ and $Q \in \Phi$.
- $P \vee Q \in \Phi$ iff $P \in \Phi$ or $Q \in \Phi$.
- $P \Rightarrow Q \in \Phi$ iff $Q \in \Phi$ whenever $P \in \Phi$.
- $P \Leftrightarrow Q \in \Phi$ iff $P \in \Phi$ iff $Q \in \Phi$.

Lemma 4.28 *Saturated sets are satisfiable.*

Proof: Let Φ be a saturated set of propositions and let $\mathcal{I}_\Phi$ be an interpretation that maps an atom to 1 if that atom is in Φ, and to 0 otherwise. Now pick an arbitrary proposition P. A straightforward induction on the structure of P will show that $\mathcal{I}_\Phi \models P$ iff $P \in \Phi$. It follows that $\mathcal{I}_\Phi$ is a model of Φ. ■

Let us say that a set Φ is *inconsistent* if $\Phi \vdash_{\mathcal{CND}}$ **false**. If that is not the case, we will call Φ *consistent*. The following provides an alternative characterization of inconsistency:

Lemma 4.29 *Φ is inconsistent iff $\Phi \vdash_{\mathcal{CND}} P$ for every proposition P, i.e., iff "everything is provable from Φ".*

Proof: If Φ is inconsistent then $\Phi \vdash_{\mathcal{CND}}$ **false**, i.e., there is a D such that

$$\beta \vdash D \leadsto \textbf{false} \tag{4.25}$$

for some $\beta \subseteq \Phi$. Pick any proposition P and let D' be the deduction

D;
suppose-absurd $\neg P$ **in**
  false;
double-negation $\neg\neg P$

From 4.25 and the semantics of $\mathcal{CND}$ it follows that $\beta \vdash D' \leadsto P$, and thus $\Phi \vdash_{\mathcal{CND}} P$. The converse direction is trivial. ■

Another useful description is the following:

Lemma 4.30 *Φ is inconsistent iff there is a P such that $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} \neg P$.*

Proof: One direction follows from Lemma 4.29. In the converse direction, suppose there is a P such that $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} \neg P$, so that $\beta_1 \vdash D_1 \leadsto P$ and $\beta_2 \vdash D_2 \leadsto \neg P$ for some deductions D_1, D_2 and subsets β_1, β_2 of Φ. Letting $\beta = \beta_1 \cup \beta_2$, the Dilution Lemma gives

$$\beta \vdash D_1 \leadsto P \quad \text{and} \quad \beta \vdash D_2 \leadsto \neg P. \tag{4.26}$$

Setting

$$D = D_1; D_2; \textbf{absurd } P, \neg P$$

4.26, $[R_5]$, and the Dilution Lemma give $\beta \vdash D \leadsto$ **false**, therefore $\Phi \vdash_{\mathcal{CND}}$ **false** and Φ is inconsistent. ■

Lemma 4.31 *If $\Phi \cup \{P\} \vdash_{\mathcal{CND}} Q$ then $\beta \cup \{P\} \vdash D \leadsto Q$ for some D and $\beta \subseteq \Phi$.*

Proof: The supposition $\Phi \cup \{P\} \vdash_{\mathcal{CND}} Q$ means that there is a D and a $\beta_0 \subseteq \Phi \cup \{P\}$ such that

$$\beta_0 \vdash D \rightsquigarrow Q. \tag{4.27}$$

Set $\beta = \beta_0 - \{P\}$, so that $\beta \subseteq \Phi$. There are two possible cases: either $P \in \beta_0$ or not. In the first case we have $\beta \cup \{P\} = \beta_0$, thus 4.27 yields $\beta \cup \{P\} \vdash D \rightsquigarrow Q$. On the other hand, if $P \notin \beta_0$ then $\beta = \beta_0$ and 4.27 becomes $\beta \vdash D \rightsquigarrow Q$; so, by dilution, $\beta \cup \{P\} \vdash D \rightsquigarrow Q$. ∎

Lemma 4.32 *If $\Phi \cup \{P_1\} \vdash_{\mathcal{CND}} P_2$ then $\Phi \vdash_{\mathcal{CND}} P_1 \Rightarrow P_2$.*

Proof: Immediate from Lemma 4.31 and Lemma 4.2. ∎

Lemma 4.33 *If $\Phi \cup \{P\} \vdash_{\mathcal{CND}} Q$ and $\Phi \cup \{P\} \vdash_{\mathcal{CND}} \neg Q$ then $\Phi \vdash_{\mathcal{CND}} \neg P$.*

Proof: By Lemma 4.31, the suppositions $\Phi \cup \{P\} \vdash_{\mathcal{CND}} Q$ and $\Phi \cup \{P\} \vdash_{\mathcal{CND}} \neg Q$ mean that there are deductions D_1, D_2 and subsets $\beta_1 \subseteq \Phi$, $\beta_2 \subseteq \Phi$, such that

$$\beta_1 \cup \{P\} \vdash D_1 \rightsquigarrow Q \quad \text{and} \quad \beta_2 \cup \{P\} \vdash D_2 \rightsquigarrow \neg Q.$$

Thus the Dilution Lemma gives

$$\beta_1 \cup \beta_2 \cup \{P\} \vdash D_1 \rightsquigarrow Q \tag{4.28}$$

and

$$\beta_1 \cup \beta_2 \cup \{P\} \vdash D_2 \rightsquigarrow \neg Q. \tag{4.29}$$

Now set

$$D \;=\; \begin{array}{l} \textbf{suppose-absurd } P \textbf{ in} \\ \quad \textbf{begin} \\ \quad\quad D_1; \\ \quad\quad D_2; \\ \quad\quad \textbf{absurd } Q, \neg Q \\ \quad \textbf{end} \end{array}$$

By 4.28, 4.29, and the semantics of $\mathcal{CND}$ we get

$$\beta_1 \cup \beta_2 \vdash D \rightsquigarrow \neg P$$

thus $\Phi \vdash_{\mathcal{CND}} \neg P$. ∎

The following will come handy in the completeness proof:

Lemma 4.34 *$\Phi \vdash_{\mathcal{CND}} P$ iff $\Phi \cup \{\neg P\}$ is inconsistent. Equivalently, $\Phi \vdash_{\mathcal{CND}} \neg P$ iff $\Phi \cup \{P\}$ is inconsistent.*

Proof: If $\Phi \vdash_{\mathcal{CND}} P$ then, by monotonicity, $\Phi \cup \{\neg P\} \vdash_{\mathcal{CND}} P$, while, by reflexivity,

$$\Phi \cup \{\neg P\} \vdash_{\mathcal{CND}} \neg P.$$

Therefore, $\Phi \cup \{\neg P\}$ is inconsistent. Conversely, if $\Phi \cup \{\neg P\}$ is inconsistent then

$$\Phi \cup \{\neg P\} \vdash_{\mathcal{CND}} Q \quad \text{and} \quad \Phi \cup \{\neg P\} \vdash_{\mathcal{CND}} \neg Q$$

for some Q, hence, by Lemma 4.33, $\Phi \vdash_{\mathcal{CND}} \neg\neg P$, and thus $\Phi \vdash_{\mathcal{CND}} P$ by Lemma 4.14. The second equivalence is proved in a similar manner. ■

If Φ is consistent and is not properly contained in any consistent Ψ then we will say that Φ is *maximally consistent*. In other words, Φ is maximally consistent iff for every proposition P that is not in Φ, the set $\Phi \cup \{P\}$ is inconsistent.

Lemma 4.35 *Maximally consistent sets are deductively closed, i.e., if Φ is maximally consistent and $\Phi \vdash_{\mathcal{CND}} P$ then $P \in \Phi$.*

Proof: Suppose Φ is maximally consistent and $\Phi \vdash_{\mathcal{CND}} P$. If $P \notin \Phi$ then $\Phi \cup \{P\}$ is inconsistent, thus $\Phi \vdash_{\mathcal{CND}} \neg P$ by Lemma 4.34, and since we also have $\Phi \vdash_{\mathcal{CND}} P$, Φ must be inconsistent—a contradiction. Consequently, we must have $P \in \Phi$. ■

Theorem 4.36 *Maximally consistent sets are saturated.*

Proof: Suppose that Φ is maximally consistent. We verify each of the five requisite properties:

(a) *$P \in \Phi$ iff $\neg P \notin \Phi$*: If both $P \in \Phi$ and $\neg P \in \Phi$ then $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} \neg P$, contradicting our assumption that Φ is consistent. On the other hand, suppose that neither $P \in \Phi$ nor $\neg P \in \Phi$. Then $\Phi \cup \{P\}$ is inconsistent (since Φ is maximally consistent), hence $\Phi \vdash_{\mathcal{CND}} \neg P$ (Lemma 4.34), hence $\neg P \in \Phi$ (since Φ is deductively closed, Lemma 4.35), contradicting our assumption that $\neg P \notin \Phi$. Therefore, exactly one of $P, \neg P$ must be in Φ.

(b) *$P \wedge Q \in \Phi$ iff $P \in \Phi$ and $Q \in \Phi$*: Suppose $P \wedge Q \in \Phi$. Then $\Phi \vdash_{\mathcal{CND}} P \wedge Q$, and Lemma 4.14 yields $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} Q$. But Φ is deductively closed, so $P \in \Phi$ and $Q \in \Phi$. Conversely, suppose $P \in \Phi$ and $Q \in \Phi$. Then $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} Q$, thus $\Phi \vdash_{\mathcal{CND}} P \wedge Q$ by Lemma 4.14, and $P \wedge Q \in \Phi$ by Lemma 4.35.

(c) $P \vee Q \in \Phi$ *iff* $P \in \Phi$ *or* $Q \in \Phi$: In one direction, suppose $P \in \Phi$. Then

$$\Phi \vdash_{\mathcal{CND}} P$$

and, by Lemma 4.14, $\Phi \vdash_{\mathcal{CND}} P \vee Q$, so $P \vee Q \in \Phi$ since Φ is deductively closed. A symmetric argument will show that $P \vee Q \in \Phi$ whenever $Q \in \Phi$. In the converse direction, suppose $P \vee Q \in \Phi$ while neither $P \in \Phi$ nor $Q \in \Phi$. Then $\neg P \in \Phi, \neg Q \in \Phi$ (by part (a)), hence $\Phi \vdash_{\mathcal{CND}} \neg P$, $\Phi \vdash_{\mathcal{CND}} \neg Q$, hence $\Phi \vdash_{\mathcal{CND}} \neg P \wedge \neg Q$ and $\Phi \vdash_{\mathcal{CND}} \neg(P \vee Q)$ by Lemma 4.14. But we also have $\Phi \vdash_{\mathcal{CND}} P \vee Q$ (since we are assuming $P \vee Q \in \Phi$), hence Φ is inconsistent, contradicting our assumption to the contrary. Therefore, one of $P \in \Phi$, $Q \in \Phi$ must hold.

(d) $P \Rightarrow Q \in \Phi$ *iff* $Q \in \Phi$ *whenever* $P \in \Phi$: Suppose first that $P \Rightarrow Q \in \Phi$, so that $\Phi \vdash_{\mathcal{CND}} P \Rightarrow Q$. Then if $P \in \Phi, \Phi \vdash_{\mathcal{CND}} P$ and

$$\Phi \vdash_{\mathcal{CND}} Q$$

by Lemma 4.14, so $Q \in \Phi$ by Lemma 4.35. Conversely, suppose $Q \in \Phi$ whenever $P \in \Phi$, and yet $P \Rightarrow Q \notin \Phi$. Then $\Phi \cup \{P\}$ contains Q, hence $\Phi \cup \{P\} \vdash_{\mathcal{CND}} Q$, and $\Phi \vdash_{\mathcal{CND}} P \Rightarrow Q$ by Lemma 4.32. Thus $P \Rightarrow Q \in \Phi$ by Lemma 4.35, and we have a contradiction.

(e) $P \Leftrightarrow Q \in \Phi$ *iff* $P \in \Phi$ *iff* $Q \in \Phi$: Suppose $P \Leftrightarrow Q \in \Phi$. Then

$$\Phi \vdash_{\mathcal{CND}} P \Leftrightarrow Q \quad \text{and} \quad \text{(a)}\, \Phi \vdash_{\mathcal{CND}} P \Rightarrow Q,\ \text{(b)}\, \Phi \vdash_{\mathcal{CND}} Q \Rightarrow P$$

(by Lemma 4.14), so if $P \in \Phi$ then $\Phi \vdash_{\mathcal{CND}} P$ and $\Phi \vdash_{\mathcal{CND}} Q$ (from (a) and Lemma 4.14), thus $Q \in \Phi$ by Lemma 4.35. By parity of reasoning with (b), if $Q \in \Phi$ then $P \in \Phi$. Therefore, $P \in \Phi$ iff $Q \in \Phi$. Conversely, suppose that $P \in \Phi$ iff $Q \in \Phi$. By (4), we get $P \Rightarrow Q \in \Phi$ and $Q \Rightarrow P \in \Phi$; hence $\Phi \vdash_{\mathcal{CND}} P \Rightarrow Q$, $\Phi \vdash_{\mathcal{CND}} Q \Rightarrow P$, and Lemma 4.14 gives $\Phi \vdash_{\mathcal{CND}} P \Leftrightarrow Q$. Thus, by Lemma 4.35, $P \Leftrightarrow Q \in \Phi$.

Both the case analysis and the proof are now complete. ■

Lemma 4.37 (Lindebaum's Lemma) *Every consistent set Φ is contained in some maximally consistent set Ψ.*

Proof: Suppose that Φ is consistent. We will demonstrate the existence of a maximally consistent set Φ^* such that $\Phi^* \supseteq \Phi$. First, let $P_1, P_2, P_3, \ldots$ be an arbitrary enumeration of all propositions (so that every proposition appears at least once in this list). Define a sequence of sets $\Phi_0, \Phi_1, \Phi_2, \ldots$ as:

$$\begin{aligned}\Phi_0 &= \Phi\\ \Phi_{i+1} &= \begin{cases}\Phi_i \cup \{P_{i+1}\} & \text{if } \Phi_i \cup P_{i+1} \text{ is consistent}\\ \Phi_i & \text{otherwise}\end{cases}\end{aligned}$$

and set

$$\Phi^* = \bigcup_{i \in N} \Phi_i.$$

The reader will verify that

(a) $\Phi_i \subseteq \Phi_{i+j}$ for all $i, j \in N$; and

(b) every Φ_i is consistent, $i \in N$.

We will now prove that Φ^* is maximally consistent. For consistency, suppose, by way of contradiction, that Φ^* is inconsistent, so that $\beta \vdash D \rightsquigarrow \textbf{false}$ for some D and

$$\beta = \{Q_1, \ldots, Q_k\} \subseteq \Phi^*.$$

Since $\Phi^* = \Phi_0 \cup \Phi_1 \cup \Phi_2 \cup \cdots$, for each $j = 1, \ldots, k$ we must have

$$Q_j \in \Phi_{i_j} \tag{4.30}$$

for some i_j. Set $n = \max\{i_1, \ldots, i_k\}$. From (a) and 4.30 it follows that $Q_j \in \Phi_n$ for every $j = 1, \ldots, k$, which is to say $\beta \subseteq \Phi_n$. But this means that $\Phi_n \vdash_{\mathcal{CND}} \textbf{false}$, contradicting (b), the fact that every Φ_i is consistent. Therefore, Φ^* must be consistent.

Next, let Q be any proposition not in Φ^*. Then $Q \notin \Phi_i$ for all $i \in N$. Now since every proposition appears in the list $P_1, P_2, P_3, \ldots$, we must have $Q = P_{n+1}$ for some $n \in N$. If $\Phi_n \cup \{Q\}$ were consistent we would have $Q \in \Phi_{n+1}$, but $Q \notin \Phi_{n+1}$, hence $\Phi_n \cup \{Q\}$ is inconsistent; and since $\Phi_n \subseteq \Phi^*$, $\Phi^* \cup \{Q\}$ is inconsistent. This shows that Φ^* is maximally consistent and concludes the argument since $\Phi^* \supseteq \Phi$. ■

Theorem 4.38 *Consistent sets are satisfiable.*

Proof: If Φ is consistent then, by the previous lemma, there is a maximally consistent Ψ such that $\Phi \subseteq \Psi$. By Theorem 4.36 and Lemma 4.28, there exists an interpetation that satisfies every member of Ψ, and thus every member of Φ as well, therefore Φ is satisfiable. ■

$$\frac{}{\beta \vdash P \rightsquigarrow (P, \beta \cup \{P\})} \; [AR_1]$$
for any $P \in \beta \cup \{\mathbf{true}, \neg\mathbf{false}\}$

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow (Q, \beta')}{\beta \vdash \mathbf{assume}\ P\ \mathbf{in}\ D \rightsquigarrow (P \Rightarrow Q, \beta \cup \{P \Rightarrow Q\})} \; [AR_2]$$

$$\frac{\beta \vdash D_1 \rightsquigarrow (P_1, \beta_1) \qquad \beta_1 \vdash D_2 \rightsquigarrow (P_2, \beta_2)}{\beta \vdash D_1; D_2 \rightsquigarrow (P_2, \beta_2)} \; [AR_3]$$

Figure 4.6: Alternative semantics for $\mathcal{CND}$.

Theorem 4.39 (Completeness of $\mathcal{CND}$) *If* $\Phi \models P$ *then* $\Phi \vdash_{\mathcal{CND}} P$.

Proof: If $\Phi \models P$ then $\Phi \cup \{\neg P\}$ is unsatisfiable, hence by Theorem 4.38, $\Phi \cup \{\neg P\}$ is inconsistent. Therefore, by Lemma 4.34, $\Phi \vdash_{\mathcal{CND}} P$. ∎

Corollary 4.40 *Every tautology is a theorem of* $\mathcal{CND}$*, i.e.,* $\emptyset \vdash_{\mathcal{CND}} P$ *whenever* $\models P$.

4.7 Variations

A number of variations on the core semantics are possible. For instance, one might argue that the reason why associativity fails for deduction composition is that our language is not sufficiently imperative. Semicolons and **begin-end** pairs aside, we are essentially taking a functional viewpoint:[4] we are treating a deduction D as something to be *evaluated* in a given assumption base, in order to produce a proposition. So, denotationally, the meaning of D is a function from assumption bases to propositions. $\mathcal{CND}$ deductions are thus analogous to side-effect-free expressions in imperative programming languages, the meaning of such an expression being a function that takes a state and produces a value (the analogy relating states to assumption bases and values to propositions).

One might take a more imperative approach by viewing a deduction D as something to be *executed*—rather than evaluated—in a given assumption base. Denotationally,

[4] This will be borne out in the sequel by the ease with which we will be able to desugar $\mathcal{CND}$ into the side-effect-free $\lambda\mu$-calculus.

the meaning of D would then be a function that takes an assumption base β and produces a *pair* (P, β') consisting of a proposition P (the conclusion of D) and another assumption base β'. Insofar as such a sheme is to be of practical value for customary logic, the assumption base β' should incorporate some or all of the propositions that were derived in the course of D, *in addition* to the final conclusion P. But although the practical intention behind such semantics would be for the final assumption base β' to be a superset of the original β, this is not necessary in theory, as one might conceive of situations such as arising in non-monotonic logics where β' does not properly contain β. Under this viewpoint, then, deductions may be regarded as analogous to expressions with side effects in imperative programming languages, the meaning of such an expression being a function that maps a given state to a pair comprising a value and a state.

We propose such a semantics in Figure 4.6. The semantic judgments are now of the form $\beta \vdash D \rightsquigarrow (P, \beta')$, to be read as "executing D in β results in P and β'". The axioms for primitive deductions are straightforward and we omit them. For example, the axiom for **both** would be

$$\beta \cup \{P, Q\} \vdash \textbf{both } P, Q \rightsquigarrow (P \wedge Q, \beta \cup \{P, Q, P \wedge Q\}).$$

The deducibility relation $\vdash_{\mathcal{CND}}$ would now be defined thus: $\beta \vdash_{\mathcal{CND}} P$ iff there are D and β' such that $\beta \vdash D \rightsquigarrow (P, \beta')$. Likewise, we may define $\beta \vdash_{\mathcal{CND}} \beta'$ iff there are D and P such that $\beta \vdash D \rightsquigarrow (P, \beta')$. Both the soundness and the completeness proofs we gave in Section 4.6 would go through with minor modifications for the former relation. A stronger soundness result can be obtained via the second relation by showing that $\beta \models \beta'$ whenever $\beta \vdash_{\mathcal{CND}} \beta'$. This subsumes the conclusion's soundness (i.e., that $\beta \models P$ whenever $\beta \vdash_{\mathcal{CND}} P$), since we have $P \in \beta'$ whenever $\beta \vdash D \rightsquigarrow (P, \beta')$. Further, we can prove that assumption bases grow monotonically during evaluation in the following sense:

$$\text{if} \quad \beta \vdash_{\mathcal{CND}} \beta' \quad \text{then} \quad \beta \subseteq \beta'.$$

Observational equivalence with respect to a fixed β would be defined in essentially the same way: $D_1 \approx_\beta D_2$ iff, for all P and β',

$$\beta \vdash D_1 \rightsquigarrow (P, \beta') \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow (P, \beta').$$

General observational equivalence $\approx$ is defined as before, by universally quantifying over the subscripted β. Composition would then be provably associative:

$$D_1; (D_2; D_3) \approx (D_1; D_2); D_3.$$

For evidence of this equality, the reader should try evaluating the two deductions of our earlier counter-example

$$D_1 = \textbf{double-negation } \neg\neg A; \textbf{begin true}; A \textbf{ end}$$

and

$$D_2 = \textbf{begin double-negation } \neg\neg A; \textbf{true}; \textbf{end}; A$$

in the assumption base $\{\neg\neg A\}$. Unlike before, the end results in both cases will now be seen to be identical: the pair $(A, \{\neg\neg A, A, \textbf{true}\})$. The reader should try to determine whether other identities continue to hold, e.g., whether it is still the case that

$$\textbf{assume } P \textbf{ in } (D_1; D_2) \approx D_1; \textbf{assume } P \textbf{ in } D_2$$

whenever $P \notin FA(D_1)$. Finally, we remark that a straightforward analogue of the Strictness Coincidence Lemma is obtainable under these semantics, and that the decision problem $D_1 \approx D_2$ remains effectively solvable.

Other variations are possible; for instance, hypothetical deductions could be instrumented to convey more information as follows:

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow (Q, \beta')}{\beta \vdash \textbf{assume } P \textbf{ in } D \rightsquigarrow (P \Rightarrow Q, \{P \Rightarrow Q \mid Q \in \beta'\})}$$

It seems, however, that the two most natural semantic models are the ones discussed in this chapter (Figure 4.2 and Figure 4.6). In particular, most of the nice properties obtainable for these two models are delicate in that they appear to be highly sensitive to modifications (e.g., with the above semantics for hypothetical deductions it would no longer be true that

$$\textbf{assume } P \textbf{ in } (D_1; D_2) \approx D_1; \textbf{assume } P \textbf{ in } D_2$$

whenever $P \notin FA(D_1)$). The choice between the various alternatives is likely to affect the experience of writing deductions in the language, as well as the proof theory of the language, and raise unique implementation issues (for instance, the semantics we have proposed in this section are clearly more expensive to implement than the earlier version); but the important metatheoretical properties—compactness, soundness, and completeness—should be left intact. A less fundamental change but one with a dramatic practical impact results from the incorporation of naming, which leads to two distinct notions of scope—assumption scope and variable scope—and paves the way for deductive abstraction via methods and higher-order “proof programming”. These issues will be explored in Chapter 8.

4.8 Composition graphs and multigraphs

In this section we study the idea of proof composition (exemplified in $\mathcal{CND}$ via the operator ;) in an abstract setting. The concepts we introduce here will pertain to some of the optimizing transformations we consider in the next chapter, but they are also of interest in their own right.

For the sake of generality our discussion will not employ any $\mathcal{CND}$-specific concepts. Rather, we will assume we have two abstract notions of *sentence* and *proof*. We will not be interested in analyzing these notions as we did for $\mathcal{CND}$; rather, we will take them as given. All we will assume is the existence of two computable functions: for each proof $\mathfrak{D}$, $\mathit{Strict}(\mathfrak{D})$ will return a set of sentences (intuitively, the sentences which are strictly "needed" or "used" in $\mathfrak{D}$); and $\mathit{Con}(\mathfrak{D})$ will return a sentence which we will call the *conclusion* of $\mathfrak{D}$. We will use the letters s, t, u, v for sentences and $\mathfrak{D}$ for proofs. The analogy that should be kept in mind in connection with $\mathcal{CND}$ is:

$$\begin{array}{rcl} \text{sentence } u & \equiv & \text{proposition } P \\ \text{proof } \mathfrak{D} & \equiv & \text{deduction } D \\ \mathit{Strict}(\mathfrak{D}) & \equiv & \mathit{FA}(D) \\ \mathit{Con}(\mathfrak{D}) & \equiv & \mathcal{C}(D) \end{array}$$

By a proof *composition* we will mean simply a finite sequence $C = \mathfrak{D}_1, \ldots, \mathfrak{D}_n$. For any such composition, we define a directed graph $(V_C, \twoheadrightarrow_C)$ where V_C is the set of *nodes* and $\twoheadrightarrow \subseteq V_C^2$ is a binary relation on V_C:

$$\begin{array}{rcl} V_C & = & \displaystyle\bigcup_{i=1}^{n} [\mathit{Strict}(\mathfrak{D}_i) \cup \{\mathit{Con}(\mathfrak{D}_i)\}] \\ \twoheadrightarrow_C & = & \displaystyle\bigcup_{i=1}^{n} \left[\bigcup_{u \in \mathit{Strict}(\mathfrak{D}_i)} \{(\mathit{Con}(\mathfrak{D}_i), u)\} \right] \end{array}$$

Thus the nodes are sentences and the relation $\twoheadrightarrow$ hooks conclusions to premises. Implicit in the above equations is an algorithm for constructing $\twoheadrightarrow_C$, namely, for $i = 1, \ldots, n$ compute the set $\mathit{Strict}(\mathfrak{D}_i)$, and for each u in that set add the pair $(\mathit{Con}(\mathfrak{D}_i), u)$ to $\twoheadrightarrow_C$. We will call $\twoheadrightarrow_C$ the *dependency kernel* of C, and if $u \twoheadrightarrow_C v$ we will say that u *depends* on v (in C). Finally, we define the *dependency relation* (or "dependency graph") of C, denoted $\twoheadrightarrow_C^+$, as the transitive closure of $\twoheadrightarrow$. If $u \twoheadrightarrow_C^+ v$ we will say that u *transitively depends* on v. Whenever it is immaterial, the subscript C will be dropped.

For a $\mathcal{CND}$ example, and with the aforementioned analogy in mind, consider the composition

double-negation $\neg\neg A$;
modus-ponens $A \Rightarrow B \wedge C, A$;
left-and $B \wedge C$;

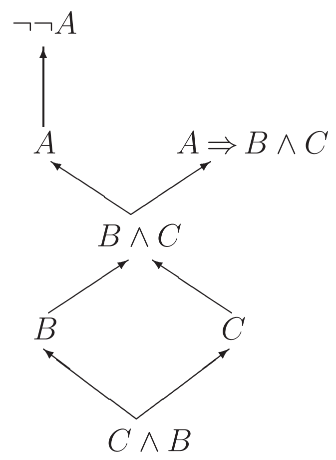


Figure 4.7: The dependency kernel of an $\mathcal{CND}$ composition.

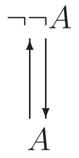
right-and $B \wedge C$;
both C, B

which derives $C \wedge B$ from $A \Rightarrow B \wedge C$ and $\neg\neg A$. If we construct the dependency kernel of this composition in stages, here are the pairs that would be added at each step:

$$(A, \neg\neg A)$$
$$(B \wedge C, A \Rightarrow B \wedge C), (B \wedge C, A)$$
$$(B, B \wedge C), (C, B \wedge C)$$
$$(C \wedge B, B), (C \wedge B, C)$$

Graphically, the resulting relation $\twoheadrightarrow$ is shown in Figure 4.7.

The $\twoheadrightarrow_C$–minimal elements will be called the *premises* of C. That is, $u \in V_C$ is a premise of C iff there is no v such that $u \twoheadrightarrow_C v$, i.e., iff u is not derived from other sentences. Nodes which are not premises will be called *intermediate conclusions*, or *lemmas*. On the assumption that *Strict*($\mathfrak{D}$) is always non-empty, $\twoheadrightarrow$ will always have at least one intermediate conclusion. The same is not true for premises, if $\twoheadrightarrow$ has cycles. For example, the dependency kernel of $D_1; D_2$, where $D_1 =$ **double-negation** $\neg\neg A$ and $D_2 =$ **suppose-absurd** $\neg A$ **in absurd** $A, \neg A$ is



which has no premises because A and $\neg\neg A$ are derived from each other.

Clearly, we should desire $\twoheadrightarrow$ to be acyclic. This is tantamount to requiring $\twoheadrightarrow^+$ to be irreflexive; and since $\twoheadrightarrow^+$ is also transitive (by definition), and assymetry is entailed by the combination of irreflexivity and transitivity, we conclude that $\twoheadrightarrow$ is acyclic iff $\twoheadrightarrow^+$ is a strict partial order. In that case, then, the graph of $\twoheadrightarrow^+$ is a Hasse diagram, i.e., a dag. If we think of premises as axioms then the absence of cycles has the aesthetically pleasing consequence that every intermediate conclusion ultimately depends on some axiom. Formally: for every lemma u there is an axiom (premise) v such that $u \twoheadrightarrow^+ v$.

Another requirement related to the utility principle of $\mathcal{CND}$ (Section 5.2) is that $\twoheadrightarrow^+$ should be *connected*. Maximally connected subgraphs of $\twoheadrightarrow$ represent extraneous threads of deduction that are unrelated to one another. In fact we will replace the connectivity requirement by a stronger condition: $\twoheadrightarrow^+$ should be *pointed*, i.e., it should have a *top* (greatest) element. Clearly, this condition implies connectivity. Intuitively, it means that the composition has a unique goal (conclusion), and that every premise and intermediate conclusion is used in order to arrive at that conclusion.

In summary, let us call a composition C *proper* iff $\twoheadrightarrow_C^+$ is a strict partial order with a top element. Observe that proper compositions are themselves composable: we may view a proper $C = \mathfrak{D}_1, \ldots, \mathfrak{D}_n$ as a proof, by definining $\mathit{Strict}(C)$ as the set of premises of C, and $\mathit{Con}(C)$ as its top element.

Dependency graphs carry enough information to be useful for a number of simple analyses of proof compositions, but their low level of granularity—which is the source of their simplicity—makes them unsuitable for answering some interesting questions. For instance, dependency graphs cannot capture the idea of repetitions. As an example, the dependency graph of the composition

$$\textbf{left-and } A \wedge B; \textbf{double-negation } \neg\neg A$$

is

$$\begin{tikzcd} A \wedge B & & \neg\neg A \\ & A \arrow[ul] \arrow[ur] & \end{tikzcd}$$

This is a pointed strict poset, and for all we know it could represent a single-element composition $C = D_1$ with $\mathit{Strict}(D_1) = \{A \wedge B, \neg\neg A\}$ and $\mathit{Con}(D_1) = A$. There is no way to tell that the underlying composition consisted of two deductions, each with one premise and both with the same conclusion. As another example, consider the composition $D = D_1; \ldots; D_{100}$, where $D_i = \textbf{double-negation } \neg\neg A$ for $i = 1, \ldots, 100$. The dependency graph of this composition is simply

$$\begin{tikzcd} \neg\neg A \\ A \arrow[u] \end{tikzcd}$$

which clearly does not express the multiple replication of effort present in D.

Another problem with dependency graphs is that they are adirectional, in that they fail to capture the left-to-right directionality of compositions. For example, both

$$\textbf{double-negation}\ \neg\neg A; \textbf{both}\ A, A$$

and

$$\textbf{both}\ A, A; \textbf{double-negation}\ \neg\neg A$$

have the same dependency graph:

$$\begin{tikzcd} \neg\neg A \\ A \arrow[u] \\ A \wedge A \arrow[u] \end{tikzcd}$$

This can be an advantage if we wish to discover the essential dependencies without regard to precedence, but oftentimes we want the ordering of a composition to be reflected in its graph.

We can ameliorate these shortcomings by enriching the basic model with a finer level of detail. One way to do this is to work with certain kinds of *multigraphs* instead of simple graphs. In particular, we may label each dependency edge with the (index of the) deduction of which it is a part. Formally, we define the *dependency multigraph* of $C = D_1, \ldots, D_n$ as a pair $(V_C, \Rrightarrow_C)$, where V_C is as before and $\Rrightarrow_C \subseteq V_C \times V_C \times \{1, \ldots, n\}$ is a ternary relation of triples (u, v, i), where i can be seen as a label for the pair (u, v). The definition is:

$$\Rrightarrow = \bigcup_{i=1}^{n} \left[\bigcup_{v \in Strict(D_i)} (Con(D_i), v, i) \right]$$

We write $u \Rrightarrow_i v$ to mean that $(u, v, i) \in \Rrightarrow$. Thus now we may have $u \Rrightarrow_i v$ and $u \Rrightarrow_j v$ for $i \neq j$, whereas before we either had $u \twoheadrightarrow v$ or not. Specifically, the relation between $\twoheadrightarrow$ and $\Rrightarrow$ is: $u \twoheadrightarrow v$ iff $u \Rrightarrow_i v$ for some $i \in \{1, \ldots, n\}$.

As an example, constructing the dependency multigraph of

$$\textbf{left-and}\ A \wedge B; \textbf{double-negation}\ \neg\neg A$$

in two stages would result in

$$\{(A, A \wedge B, 1), (A, \neg\neg A, 2)\}$$

which may be depicted graphically as

$$\begin{tikzcd}[column sep=small] A \wedge B & & \neg\neg A \\ & A \arrow[ul, "1"] \arrow[ur, "2"'] & \end{tikzcd}$$

indicating that A is derived from $A \wedge B$ via the first deduction, and also from $\neg\neg A$ via the second deduction. Likewise, the dependency multigraph of $D_1; \ldots; D_{100}$ with each $D_i =$ **double-negation** $\neg\neg A$ would be

$$\begin{tikzcd} \neg\neg A \\ A \arrow[u, shift left=2, "100"'] \arrow[u, shift right=2, "1"] \arrow[u, phantom, "\cdots"] \end{tikzcd}$$

while those of

$$\textbf{double-negation } \neg\neg A; \textbf{both } A, A$$

and

$$\textbf{both } A, A; \textbf{double-negation } \neg\neg A$$

would respectively be

$$\begin{tikzcd} \neg\neg A \\ A \arrow[u, "1"'] \\ A \wedge A \arrow[u, "2"'] \end{tikzcd} \qquad \text{and} \qquad \begin{tikzcd} \neg\neg A \\ A \arrow[u, "2"'] \\ A \wedge A \arrow[u, "1"'] \end{tikzcd}$$

We can now modify our definition of proper deductions by requiring (a) $i = j$ whenever $u \Rrightarrow_i v$ and $u \Rrightarrow_j v$; and (b) $i > j$ whenever $u_1 \Rrightarrow_i u_2 \Rrightarrow_j u_3$. Enforcing (a) and (b) would be one way of addressing the issues of repetitions and directionality, respectively.

4.9 Towards proof programming in functional style

As it stands currently, $\mathcal{CND}$ delivers on almost every requirement we singled out in the introduction (restricted to the admittedly narrow domain of propositional logic): it is readable and writable, it has a formal syntax and semantics that support a rigorous theory of proof equivalence and optimization (to be developed in the upcoming chapter), it provides intuitive and efficient proof checking, and has a fully worked out metatheory. The one—important—omission is an abstraction mechanism that would allow for parameterized proofs and task decomposition. We will develop such a mechanism in the general setting of the $\lambda\mu$-calculus in Chapter 8, and will adapt it to $\mathcal{CND}$ in particular in Section 9.4, so there is no need to duplicate that effort here. What we will briefly do below is outline some of the major issues involved in that endeavor, so that the reader will be able to appreciate the main challenges.

It is clear that the first step towards an abstraction mechanism must be to introduce naming. In the setting of $\mathcal{CND}$, a first approximation to this step is allowing propositions to contain variables. Let us assume that we have an infinite supply of *propositional variables* $p, q, r, \ldots$, that can be be used to range over propositions (these variables must be distinct from the atoms $A, B, C, \ldots$). We can now define *propositional expressions*, or simply *expressions*, as follows:

$$E ::= p \mid A \mid \mathbf{true} \mid \mathbf{false} \mid \neg E \mid E_1 \wedge E_2 \mid E_1 \vee E_2 \mid E_1 \Rightarrow E_2 \mid E_1 \Leftrightarrow E_2.$$

Examples are $p \Rightarrow \neg A, q, (B \wedge \neg p) \vee r$, etc. Thus propositional expressions have the same structure as propositions, but variables are also allowed at the base level. We will continue to use the upper-case letters P, Q, R, ..., for propositions, which are defined simply as ground propositional expressions (i.e., such that do not contain any variables).

We may now define deductions with the following abstract syntax:

$$D \;\; ::= \;\; E \mid \textit{Prim-Rule}\ E_1, \ldots, E_n \mid \mathbf{assume}\ E\ \mathbf{in}\ D \mid \mathbf{let}\ p = D_1\ \mathbf{in}\ D_2$$

Deductions of the form E serve as claims; those of the form *Prim-Rule* E_1, ..., E_n and **assume** E **in** D are again atomic and hypothetical deductions, respectively; while composite deductions are now of the form **let** $p = D_1$ **in** D_2.

When faced with the task of giving an evaluation semantics for this language it becomes apparent that the basic judgments can no longer be of the form $\beta \vdash D \rightsquigarrow P$. The complicating factor is naming, i.e., the presence of variables. Suppose, for instance, that our deduction is the simple claim p. Clearly, we have no way of telling whether this claim holds (in the context of a given β) if we do not know what proposition p stands for. What is missing is the notion of lexical environments. In this case, an environment ρ can be understood as a computable function that maps each variable either to a proposition or to a special "unbound" token.

The new semantic judgments are of the form $\rho, \beta \vdash D \rightsquigarrow P$, to be read "In environment ρ and assumption base β, deduction D yields the conclusion P". These judgments are defined in Figure 4.8. An auxiliary type of judgment $\rho \vdash E \hookrightarrow P$ is used, which signifies that in the context of the environment ρ, E denotes the proposition P. These judgments are defined via the rule

$$\frac{}{\rho \vdash p \hookrightarrow P}$$

whenever $\rho(p) = P$

$$\frac{}{\rho, \beta \vdash \mathbf{true} \leadsto \mathbf{true}} \text{[F2]} \qquad \frac{}{\rho, \beta \vdash \neg\mathbf{false} \leadsto \neg\mathbf{false}} \text{[F2]}$$

$$\frac{\rho \vdash E \hookrightarrow P}{\rho, \beta \cup \{P\} \vdash E \leadsto P} \text{[F3]} \qquad \frac{\rho \vdash E \hookrightarrow P \qquad \rho, \beta \cup \{P\} \vdash D \leadsto Q}{\rho, \beta \vdash \mathbf{assume}\ E\ \mathbf{in}\ D \leadsto P \Rightarrow Q} \text{[F4]}$$

$$\frac{\rho, \beta \vdash D_1 \leadsto P \qquad \rho[p \mapsto P], \beta \cup \{P\} \vdash D_2 \leadsto Q}{\rho, \beta \vdash \mathbf{let}\ p = D_1\ \mathbf{in}\ D_2 \leadsto Q} \text{[F5]}$$

Figure 4.8: A first semantics for a functional-style proof language.

and rules of the form

$$\frac{\rho \vdash E_1 \hookrightarrow P_1 \qquad \rho \vdash E_2 \hookrightarrow P_2}{\rho \vdash E_1 \wedge E_2 \hookrightarrow P_1 \wedge P_2}$$

(with similar rules for $\neg$, $\vee$, etc; the constants **true** and **false** evaluate to themselves). Also, the rules for primitive deductions are straightforward and not depicted in Figure 4.8; we present Modus Ponens as an example:

$$\frac{\rho \vdash E_1 \hookrightarrow P \Rightarrow Q \qquad \rho \vdash E_2 \hookrightarrow P}{\rho, \beta \cup \{P \Rightarrow Q, P\} \vdash \mathbf{modus\text{-}ponens}\ E_1, E_2 \leadsto Q}$$

As presently formulated, this language is far from being a functional-style "proof-programming language". In the world of programming, functional languages are characterized mainly by the first-class presence of λs, i.e., by possibly anonymous functional abstractions which can be passed around and manipulated in a higher-order manner. Presently we have no comparable notion of abstraction and application. Indeed, it is not even clear what form such notions would take in a deductive setting, and how they would mesh with assumption-base semantics. So, as it stands, this language is essentially $\mathcal{CND}$ in a slightly different guise: the composition operator ; has been replaced by **let**.

However, we have taken the important first step of introducing variables, which can serve as *parameters*. That clears up the road for *abstraction by parameterization*, which is exactly what the λ achieves in the case of computation. We quote from Liskov and Guttag [35]:

> Abstraction by parameterization allows us, through the introduction of parameters, to represent a potentially infinite set of different computations

with a single program text that is an abstraction of all of them. Consider the program text

$$x * x + y * y$$

This describes a computation that adds the square of the value stored in a particular variable x to the square of the value stored in another particular variable y. The *lambda expression*

$$\lambda x, y : \mathrm{int}.(x * x + y * y)$$

on the other hand, describes the set of computations that square the value stored in some integer variable x, which we shall temporarily refer to as x, and add it to the square of the value stored in another integer variable, which we shall temporarily call y.

In a similar vein, consider a deduction containing variables:

assume $p \wedge q$ **in**
  left-and $p \wedge q$

This can be viewed as a deduction *schema* respresenting the infinite set of deductions which are obtainable by instantiating p and q with particular propositions. In this case, no matter what propositions we assign to p and q, the end result of the deduction will always be the tautology $p \wedge q \Rightarrow p$. We introduce the letter μ as the deductive counterpart of λ. Thus if D is a deduction possibly containing variables $p_1, \ldots, p_n$, then

$$\mu\, p_1, \ldots, p_n .\ D$$

will be a deduction abstraction, called a *method*, that parameterizes D over these variables.

Several questions are immediately raised. At the language level, in the abstract syntax, what kind of things will methods be? Right now there is only one syntactic category, that of deductions D. But methods are not deductions, they are abstractions thereof. A deduction is something that can be evaluated to produce a proposition (its conclusion), or maybe fail; but the evaluation of a method will clearly not produce a proposition, just as the evaluation of a λ-abstraction does not produce a number or a string. Dually, what kind of things will methods be at the semantic level? So far we only have one kind of data value—propositions. Evaluation always either produces a proposition or fails. But what kind of data value will be produced by evaluating, say,

$$\mu\, p, q\,.\, \textbf{assume}\ p \wedge q\ \textbf{in left-and}\ p \wedge q?$$

And will methods be denotable values? That is, will a method be able to take another method as an argument? If so, variables ought to range over propositions *and* methods, not just propositions. More importantly, we still don't have a dual notion of method application. How are methods to be applied? Could a method application produce another method? Should a method be able to call itself recursively? Should primitive deductions be subsumed by method applications?

There are many other issues. There is no conditional branching, for instance. A method for applying De Morgan's laws, for example, must be able to check whether its input formula has the right form and act accordingly. We cannot write such a method presently. Another problem is that the naming mechanism provided by the **let** is too rigid, in the sense that a name can be attached only to a proposition that has been deductively derived. This is dictated by the syntax of the language: a **let** binding is of the form $p = D$, and thus a variable (p) may only become bound to the result of a *deduction* (D). But this is often impractical, and is also not a good model of the way people present proofs in practice. Suppose, for instance, that β contains $\neg(A \vee B) \Rightarrow C$ and $\neg(A \vee B)$, and nothing else. Then I should be able to evaluate

let $p = A \vee B$
in
  modus-ponens $\neg p \Rightarrow C, \neg p$

in β and obtain the conclusion C. But presently this would fail because in evaluating the binding $p = A \vee B$ the proposition $A \vee B$ would be interpreted as a claim, and since $A \vee B \notin \beta$, an error would occur.

We will resolve all of these issues in the general setting of the $\lambda\mu$-calculus. In closing, however, one might ask why not use the customary notions of abstraction and application instead of inventing new ones. The answer is simple: soundness. Here is a perfectly good customary abstraction: $f = \lambda\, p\,.\,\neg p$. Here is a perfectly good application: $f(\mathbf{true})$. The result: $\neg\mathbf{true}$. What went wrong? The point is that everything that is provable is also computable, since proving is an effective process, but *not* vice versa. We have to block out those computations—effective processes—that do not represent logical derivations. If we adopted the regular notions of abstraction and application, we would have to resort to some kind of type system in order to enforce this separation. We oppose that approach because we believe that proofs and programs are sufficiently distinct to warrant a sharp differentiation at a fundamental language level (i.e., at the level of the underlying syntax and semantics), rather than at the more superficial level of a type system. The core thesis of this document is that such a differentiation is not only possible, but can be effected in a way that is both theoretically elegant and of practical value.

"These things are not for the best, nor as I think they ought to be; but still they are better than that which is downright bad."

Plautus

5 Proof optimization

5.1 Background

In this chapter we put forth an optimization procedure for $\mathcal{CND}$ deductions. The subject is clearly related to proof-tree normalization in the sense of Prawitz [59] (or alternatively, cut-elimination in sequent-based systems [29, 23]), but there are some important differences. First, Prawitz normalization is easier than $\mathcal{CND}$ optimization. In the intuitionist case, the Curry-Howard correspondence means that Prawitz normalization coincides with reduction in the simply typed λ-calculus. Accordingly, the normalization algorithm is particularly simple: keep contracting as long as there is a redex (either a β-redex or one of the form $l(\langle e_1, e_2\rangle)$, $r(\langle e_1, e_2\rangle)$, etc.). Strong normalization and the Church-Rosser property guarantee that eventually we will converge to a unique normal form. In the classical case, there is some pre-processing to be done (see Section I, Chapter III of Prawitz's book [59]) before carrying out the reductions, but they are minimal. The $\mathcal{CND}$ transformations we consider in this section are also strongly terminating, but they are much more involved.

One reason for the discrepancy is that the analysis of $\mathcal{CND}$ deductions is complicated by the presence of scope and composition: because assumptions and intermediate conclusions can have limited and arbitrarily nested scopes, it is not possible to carry out contractions in a local manner; the overall surrounding context must usually be taken into consideration. Further, the result of one transformation might affect the applicability or outcome of another transformation, so the order in which these occur is important. By contrast, in a proof tree contractions can take place locally, and in an arbitrary order. Secondly, our optimizations are more involved because they are more ambitious and produce sharper results. Prawitz's notion of normal form is very weak from the viewpoint of proof optimality. That is, it does not capture proof optimality well: a proof might be in normal form and yet be flagrantly redundant. A typical example is

$$\dfrac{\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{A}}{A \wedge A}$$

The redundancy here is the duplicate derivation of A from $A \wedge B$. Yet the proof is in Prawitz normal form, so if we conflate normalization with optimization—as is usually done [59, 34]—then we ought to identify the above proof as optimal, a curious view in light of the glaring duplication of effort. The defect ultimately stems from a more fundamental problem, the fact that proof trees (or, through the Curry-Howard correspondence, λ-calculus terms) are fundamentally misguided as a formal model of deduction (see Section 7.2.3).[1] In contradistinction, it will be seen that our optimization procedure goes beyond Prawitz's transformations by aggresively eliminating additional sources of redundancy, often resulting in dramatically simpler deductions.

Our optimization procedure consists of a series of transformations, which fall into two groups:

- *restructuring transformations*; and
- *contracting transformations*, or simply *contractions*.

Contracting transformations form the bedrock of the optimization process: they remove extraneous parts, thereby reducing the size and complexity of a deduction. Restructuring transformations simply rearrange the structure of a deduction so as to better expose contraction opportunities; in some cases they might even temporarily increase the size of a deduction.[2] Differently put, restructuring transformations constitute a kind of pre-processing intended to facilitate the contracting transformations.

Specifically, our optimization procedure *normalize* is defined as follows:

$$\mathit{normalize} = \mathit{contract} \cdot \mathit{restructure} \tag{5.1}$$

[1] We have already seen how this problem rears its head in LF, which is based on a proof-tree view of deduction.

[2] Any gratuitous expansions, however, will always be eliminated by subsequent contractions. In particular, we can prove that the size of the end result of the optimization procedure will never be larger than that of the original.

where $\cdot$ denotes ordinary function composition and

$$contract = \mathfrak{C} \cdot \mathfrak{P} \cdot \mathfrak{U} \tag{5.2}$$

while

$$restructure = inter\text{-}leave(\mathfrak{MS}, [\mathfrak{A}_3, \mathfrak{A}_2, \mathfrak{A}_1]). \tag{5.3}$$

The interleaving function is defined as:

$inter\text{-}leave(g, L) = compose^*(L)$
where
  $compose^*([]) = g$
  $compose^*(h{::}L') = g \cdot h \cdot compose^*(L')$

We will continue to define functions in this informal notation, using pattern matching, recursion, etc., in the style of languages such as ML, Haskell, or Miranda. Any reader moderately familiar with a programming language of this kind should be able to make sense of our definitions.

In the next two sections we will discuss each group of transformations in turn: first the contractions $\mathfrak{C}$, $\mathfrak{P}$, and $\mathfrak{U}$; and then the restructuring transformations $\mathfrak{MS}$, $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$.

5.2 Contracting transformations

Informally, our contracting transformations will be based on two simple principles:

Utility: *Every intermediate conclusion should be used at some later point as an argument to a primitive inference rule.*

Parsimony: *At no point should a non-trivial deduction establish something that has already been established, or something that has been hypothetically postulated.*

These principles are in turn based on the notions of *redundancies* and *repetitions*, which we will now study in detail.

5.2.1 Redundancies

Intuitively, a deduction contains redundancies if it generates conclusions which are not subsequently used. For all practical purposes, such conclusions are useless "noise". We will see that they can be systematically eliminated. Redundancy-free deductions

$$\frac{}{\vdash_{Strict} P} \qquad \frac{}{\vdash_{Strict} \textit{Prim-Rule}\ P_1, \ldots, P_n}$$

$$\frac{\vdash_{Strict} D}{\vdash_{Strict} \textbf{assume}\ P\ \textbf{in}\ D} \qquad \frac{\vdash_{Strict} D_1 \quad \vdash_{Strict} D_2 \quad \mathcal{C}(D_1) \in \textit{FA}(D_2)}{\vdash_{Strict} D_1; D_2}$$

Figure 5.1: Definition of *strict* deductions.

will be called *strict*. As a very simple example, the following deduction, which proves $A \wedge B \Rightarrow A$, is not strict:

$$\textbf{assume}\ A \wedge B\ \textbf{in begin right-and}\ A \wedge B; \textbf{left-and}\ A \wedge B; \textbf{end}$$

The redundancy here is the application of **right-and** to derive B. This is superfluous because it plays no role in the derivation of the final conclusion. We formally define the judgement $\vdash_{Strict} D$, "D is strict", in Figure 5.1. Verbally, the definition can be phrased as follows:

- *Claims and primitive deductions are always strict.*
- *A hypothetical deduction is strict if its body is strict.*
- *A composite deduction $D_1; D_2$ is strict if both D_1 and D_2 are strict, and the conclusion of D_1 is strictly used in D_2.*

The last of the above clauses is the most important one, since composition is the mechanism for linearly ordering the derivations of the various intermediate conclusions. Note that we require that $\mathcal{C}(D_1)$ be *strictly* used in D_2. Accordingly, the deduction

$$\textbf{left-and}\ A \wedge B; \textbf{assume}\ A\ \textbf{in both}\ A, A$$

is not strict: the derivation of A via **left-and** is extraneous because the only subsequent use of A, as a premise to **both** inside the **assume**, has been "buffered" by the hypothetical postulation of A.

We will now present a tranformation algorithm $\mathfrak{U}$ that converts a given deduction D into a strict deduction D'. We will prove that $\vdash_{Strict} D'$, and also that the semantics of D are conservatively preserved in the sense that $D \rightarrowtail D'$. The transformation is defined by structural recursion:

$\mathfrak{U}(\textbf{assume } P \textbf{ in } D) = \textbf{assume } P \textbf{ in } \mathfrak{U}(D)$
$$\begin{aligned}\mathfrak{U}(D_1; D_2) = \mathit{let}\ \ & D_1' = \mathfrak{U}(D_1)\\ & D_2' = \mathfrak{U}(D_2)\\ \mathit{in}\ \ & \\ & \mathcal{C}(D_1') \notin \mathit{FA}(D_2') \rightarrow D_2', D_1'; D_2'\end{aligned}$$
$\mathfrak{U}(D) = D$

We have:

Theorem 5.1 (a) $\mathfrak{U}$ *always terminates;* (b) $\mathfrak{U}(D)$ *is strict;* (c) $D \rightarrowtail \mathfrak{U}(D)$.

Proof: Termination is clear, since the size of the argument strictly decreases with each recursive call. We prove (b) and (c) simultaneously by structural induction on D.

The base cases of claims and atomic deductions are immediate. When D is of the form **assume** P **in** D_b, we have

$$\mathfrak{U}(D) = \textbf{assume } P \textbf{ in } \mathfrak{U}(D_b). \tag{5.4}$$

By the inductive hypothesis, $\mathfrak{U}(D_b)$ is strict, hence so is $\mathfrak{U}(D)$, by the definition of strictness. Further, we have $D_b \rightarrowtail \mathfrak{U}(D_b)$ by the inductive hypothesis, hence by Lemma 4.22 we get

$$\textbf{assume } P \textbf{ in } D_b \rightarrowtail \textbf{assume } P \textbf{ in } \mathfrak{U}(D_b)$$

which is to say, by virtue of 5.4, that $D \rightarrowtail \mathfrak{U}(D)$.

Finally, suppose that D is a composite deduction $D_1; D_2$ and let $D_1' = \mathfrak{U}(D_1)$, $D_2' = \mathfrak{U}(D_2)$. Either $\mathcal{C}(D_1') \in \mathit{FA}(D_2')$ or not. If yes, then $\mathfrak{U}(D) = D_1'; D_2'$, and strictness follows from the inductive hypothesis and our supposition that $\mathcal{C}(D_1') \in \mathit{FA}(D_2')$, according to the definition of $\vdash_{\mathit{Strict}}$; while $D \rightarrowtail \mathfrak{U}(D)$ in this case means $D_1; D_2 \rightarrowtail D_1'; D_2'$, which follows from the inductive hypotheses in tandem with Lemma 4.22. In contradistinction, suppose that $\mathcal{C}(D_1') \notin \mathit{FA}(D_2')$, so that $\mathfrak{U}(D) = D_2'$. Since $D = D_1; D_2 \rightarrowtail D_1'; D_2'$ follows from the inductive hypotheses and Lemma 4.22, if we can show that $D_1'; D_2' \rightarrowtail D_2'$ then $D \rightarrowtail D_2' = \mathfrak{U}(D)$ will follow from the transitivity of $\rightarrowtail$ (Lemma 4.21). We can show $D_1'; D_2' \rightarrowtail D_2'$ by using the Strictness Coincidence Lemma (Lemma 4.13). Suppose, in particular, that $\beta \vdash D_1'; D_2' \rightsquigarrow Q$, for arbitrary β and Q. By definition, this means that $\beta \vdash D_1' \rightsquigarrow P$ and

$$\beta \cup \{P\} \vdash D_2' \rightsquigarrow Q \tag{5.5}$$

for $P = \mathcal{C}(D_1')$. It follows from the assumption $P = \mathcal{C}(D_1') \notin \mathit{FA}(D_2')$ that

$$\beta \cup \{P\} \equiv_{\mathit{FA}(D_2')} \beta$$

so from 5.5 and the Strictness Coincidence Lemma we infer $\beta \vdash D_2' \rightsquigarrow Q$. We have thus shown that for any β and Q, if $\beta \vdash D_1'; D_2' \rightsquigarrow Q$ then $\beta \vdash D_2' \rightsquigarrow Q$, which is to say $D_1'; D_2' \rightarrowtail D_2'$. It follows from our earlier remarks that $D = D_1; D_2 \rightarrowtail D_2' = \mathfrak{U}(D)$. This completes the inductive argument. ■

As an illustration, suppose we wish to use the algorithm to remove redundancies from the deduction

$$D_1; D_2; \textbf{both } A, B; \textbf{left-either } A, C \tag{5.6}$$

where $\mathcal{C}(D_1) = A, \mathcal{C}(D_2) = B$. Assuming that D_1 and D_2 are already strict, the interesting reduction steps taken by the algorithm, in temporal order, may be depicted in the following loose form (where we use the arrow $\Longrightarrow$ to represent a reduction step):

1. $\textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (as $A \wedge B \notin FA(\textbf{left-either }$ A, C))
2. $D_2; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (as $\mathcal{C}(D_2) = B \notin FA(\textbf{left-either }$ A, C))
3. $D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow D_2; \textbf{left-either } A, C$ (from 1)
4. $D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow \textbf{left-either } A, C$ (from 2 and 3)
5. $D_1; D_2; \textbf{both } A, B; \textbf{left-either } A, C \Longrightarrow D_1; \textbf{left-either } A, C$ (from 4)

Thus the original deduction becomes reduced to D_1; **left-either** A, C.

5.2.2 Repetitions

The principle of utility alone cannot guarantee that a deduction will not have superfluous components. For instance, consider a slight modification of example 5.6:

$$D_1; D_2; \textbf{both } A, B; \textbf{left-and } A \wedge B \tag{5.7}$$

where again $\mathcal{C}(D_1) = A$, $\mathcal{C}(D_2) = B$. The difference with 5.6 is that the last deduction is **left-and** $A \wedge B$ instead of **left-either** A, C. In this case algorithm $\mathfrak{U}$ will have no effect because the deduction is already strict: D_1 establishes A; D_2 establishes B; then we use both A and B to obtain $A \wedge B$; and finally we use **left-and** $A \wedge B$ to get A. Thus the principle of utility is observed. The principle of parsimony, however, is clearly violated: the **left-and** deduction establishes something (A) which has already been established by D_1. For that reason, it is superfluous, and hence so are the derivations of B and $A \wedge B$.

This example illustrates what Prawitz called a *detour*: the gratuitous application of an introduction rule followed by the application of a corresponding elimination rule that gets us back to a premise which we had supplied to the introduction rule. The reason why these are detours is because elimination rules are the inverses of introduction

D_1; // proves P
$\vdots$
suppose-absurd $\neg P$ **in**
  D_2;
$\vdots$
double-negation $\neg\neg P$;
$\vdots$

(a) Detour for $\neg$.

D_1; // proves P
$\vdots$
D_2; // proves Q
$\vdots$
both P, Q;
$\vdots$
left-and $P \wedge Q$;
$\vdots$

(b) Detour for $\wedge$.

$\{D_1$; // proves $Q\}$
$\vdots$
D_2; // proves P
$\vdots$
assume P **in**
  D_3; // proves Q
$\vdots$
modus-ponens $P \Rightarrow Q, P$;
$\vdots$

(c) Detour for $\Rightarrow$.

D_1; // proves P
$\vdots$
left-either P, Q;
$\vdots$
assume P **in**
  D_2; // proves Q'
$\vdots$
assume Q **in**
  D_3; // proves Q'
$\vdots$
constructive-dilemma $P \vee Q, P \Rightarrow Q', Q \Rightarrow Q'$;
$\vdots$

(d) Detour for $\vee$.

assume P **in**
  D_1; // proves Q
$\vdots$
assume Q **in**
  D_2; // proves P
$\vdots$
equivalence $P \Rightarrow Q, Q \Rightarrow P$;
$\vdots$
left-iff $P \Leftrightarrow Q$;
$\vdots$

(e) Detour for $\Leftrightarrow$.

Figure 5.2: Prawitz-type detours for $\mathcal{CND}$.

rules. Prawitz enunciated this intuition with an informal statement that he called "the inversion principle". Figure 5.2 shows the form that Prawitz's detours take in $\mathcal{CND}$ for each of the five connectives. For $\wedge$, $\vee$, and $\Leftrightarrow$ there are twin detours, which we do not depict here, where **right-and**, **right-either**, and **right-iff** take the place of **left-and**, **left-either**, and **left-iff**, respectively. Furthermore, the detour contained in each of the threads shown in Figure 5.2 is insensitive to the ordering of most of the thread's elements: for instance, in thread (b) we may be able to swap D_1 and D_2, but this would not affect the detour; in (c) we might put D_1 immediately before the **modus-ponens**, but we would still have the same detour; and so on. So the threads of Figure 5.2 should be understood up to some permutation of the depicted elements (of course one ordering constraint that must always be respected is that elimination rules should come after introduction rules). Finally, D_1 in (c) is optional, indicated by the braces around it; we would still have essentially the same detour in the absence of D_1.

It is important to realize that Prawitz's reductions are not readily applicable in $\mathcal{CND}$. Detours may not be freely replaced by their obvious contractions; the greater context in which the subdeduction occurs will determine whether the replacement is permissible. For example, the boxed subdeduction below represents a detour, but we may not blindly simplify it because $\mathcal{C}(D_2)$, or $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$, or both, might be needed inside D':

$$\cdots; D_1; \boxed{D_2; \textbf{both } \mathcal{C}(D_1), \mathcal{C}(D_2); \textbf{left-and } \mathcal{C}(D_1) \wedge \mathcal{C}(D_2)}; \cdots D' \cdots$$

What we *can* do, however, is replace the inference **left-and** $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$ by the trivial claim $\mathcal{C}(D_1)$. A subsequent strictness analysis will determine whether $\mathcal{C}(D_2)$ and/or $\mathcal{C}(D_1) \wedge \mathcal{C}(D_2)$ are needed at any later point. If not, then we can be sure that the deductions D_2 and **both** $\mathcal{C}(D_1), \mathcal{C}(D_2)$ were indeed a detour, and algorithm $\mathfrak{U}$ will eliminate them. We will see that this simple technique of

1. replacing every deduction whose conclusion P has already been established by the trivial claim P, and then

2. removing redundancies with our utility analysis

will be sufficient for the elimination of most of the detours shown in Figure 5.2. The first step can result in a deduction with various trivial claims sprinkled throughout. This is mostly a cosmetic annoyance, but a simple contracting analysis that we will present shortly will eliminate all extraneous claims. That analysis will always be performed at the end of all other transformations in order to clean up the final result. We now present an algorithm $\mathfrak{P}$ for performing the first step of the above process:

$$
\begin{array}{l}
\mathfrak{P}(D) = RR(D, \emptyset) \\
\text{where} \\
RR(D, \Phi) = \mathcal{C}(D) \in \Phi \rightarrow \; \mathcal{C}(D), \\
\qquad\qquad\qquad\qquad\qquad \textit{match } D \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{assume } P \textbf{ in } D_b \rightarrow \textbf{assume } P \textbf{ in } RR(D_b, \Phi \cup \{P\}) \\
\qquad\qquad\qquad\qquad\qquad\quad D_1; D_2 \rightarrow \; \textit{let } D_1' = RR(D_1, \Phi) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad D_1'; RR(D_2, \Phi \cup \{\mathcal{C}(D_1')\}) \\
\qquad\qquad\qquad\qquad\qquad\quad D \rightarrow D
\end{array}
$$

The following lemma will be necessary in proving the correctness of this transformation.

Lemma 5.2 *If* $\beta \vdash RR(D, \Phi) \leadsto Q$ *then* $\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \leadsto Q$.

Proof: By structural induction on D. When D is a claim P_1 then for all Ψ,

$$RR(D, \Psi) = P_1$$

so the assumption $\beta \vdash RR(D, \Phi) \leadsto Q$ is tantamount to

$$\beta \vdash P_1 \leadsto P_1. \tag{5.8}$$

On that assumption, we need to prove that

$$\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \leadsto Q = P_1$$

i.e., $\beta \cup \{P\} \vdash P_1 \leadsto P_1$. But this follows directly from 5.8 by the monotonicity lemma.

Next, suppose that D is an atomic deduction. We distinguish two cases: either $\mathcal{C}(D) \in \Phi$ or not. In the former case we have $RR(D, \Phi) = \mathcal{C}(D)$, so the assumption $\beta \vdash RR(D, \Phi) \leadsto Q = \mathcal{C}(D)$ means that

$$\beta \vdash \mathcal{C}(D) \leadsto \mathcal{C}(D). \tag{5.9}$$

Now since we are assuming $\mathcal{C}(D) \in \Phi$, we have $\mathcal{C}(D) \in \Phi \cup \{P\}$, hence

$$RR(D, \Phi \cup \{P\}) = \mathcal{C}(D)$$

so we need to show that $\beta \cup \{P\} \vdash \mathcal{C}(D) \leadsto \mathcal{C}(D)$. But this follows from 5.9 via monotonicity. By contrast, if $\mathcal{C}(D) \notin \Phi$ then $RR(D, \Phi) = D$, so we are assuming that

$$\beta \vdash D \leadsto \mathcal{C}(D) \tag{5.10}$$

and we need to show that

$$\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \leadsto \mathcal{C}(D). \tag{5.11}$$

Now there are two subcases, namely, either $\mathcal{C}(D) = P$ or not. If $\mathcal{C}(D) = P$ then

$$RR(D, \Phi \cup \{P\}) = P$$

and 5.11 follows from reflexivity. If $\mathcal{C}(D) \neq P$ then $RR(D, \Phi \cup \{P\}) = D$ (since then $\mathcal{C}(D) \notin \Phi \cup \{P\}$), so 5.11 is tantamount to $\beta \cup \{P\} \vdash D \leadsto \mathcal{C}(D)$, which follows from 5.10 by monotonicity. This completes both case analyses.

When D is of the form **assume** P_1 **in** D_1, the assumption $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$ translates to

$$\beta \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow P_1 \Rightarrow P_2 \tag{5.12}$$

where

$$\beta \cup \{P_1\} \vdash RR(D_1, \Phi \cup \{P_1\}) \rightsquigarrow P_2. \tag{5.13}$$

Now $RR(D, \Phi \cup \{P\}) = \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\})$, so what we need to show is

$$\beta \cup \{P\} \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\}) \rightsquigarrow P_1 \Rightarrow P_2.$$

There are two cases: (a) $P = P_1$, and (b) $P \neq P_1$. In (a), the result follows from 5.12 and monotonicity, since $\Phi \cup \{P_1\} = \Phi \cup \{P, P_1\}$. If (b) holds, then, from the inductive hypothesis, 5.13 entails that

$$\beta \cup \{P\} \cup \{P_1\} \vdash RR(D_1, \Phi \cup \{P_1\} \cup \{P\}) \rightsquigarrow P_2$$

and hence

$$\beta \cup \{P\} \vdash \textbf{assume } P_1 \textbf{ in } RR(D_1, \Phi \cup \{P, P_1\}) \rightsquigarrow P_1 \Rightarrow P_2$$

which is exactly what we wanted to show.

Finally, suppose that D is of the form $D_1; D_2$ and that $\beta \vdash RR(D, \Phi) \rightsquigarrow Q$, which is to say

$$\beta \vdash D_1'; D_2' \rightsquigarrow Q \tag{5.14}$$

where

$$D_1' = RR(D_1, \Phi) \tag{5.15}$$

$$D_2' = RR(D_2, \Phi \cup \{P_1\}) \tag{5.16}$$

$$\beta \vdash D_1' \rightsquigarrow P_1 \tag{5.17}$$

and

$$\beta \cup \{P_1\} \vdash D_2' \rightsquigarrow Q. \tag{5.18}$$

We need to prove $\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow Q$, i.e.,

$$\beta \cup \{P\} \vdash D_1''; D_2'' \rightsquigarrow Q \tag{5.19}$$

where

$$D_1'' = RR(D_1, \Phi \cup \{P\}) \tag{5.20}$$

and

$$D_2'' = RR(D_2, \Phi \cup \{P\} \cup \mathcal{C}(D_1'')). \tag{5.21}$$

On the basis of 5.15 and the inductive hypothesis, 5.17 implies that

$$\beta \cup \{P\} \vdash RR(D_1, \Phi \cup \{P\}) \rightsquigarrow P_1$$

i.e.,

$$\beta \cup \{P\} \vdash D_1'' \rightsquigarrow P_1 \tag{5.22}$$

so that

$$\mathcal{C}(D_1'') = P_1. \tag{5.23}$$

Likewise, by virtue of 5.16 and the inductive hypothesis, 5.18 implies that

$$\beta \cup \{P\} \cup \{P_1\} \vdash RR(D_2, \Phi \cup \{P_1\} \cup \{P\}) \rightsquigarrow Q. \tag{5.24}$$

From 5.21 and 5.23 we get $D_2'' = RR(D_2, \Phi \cup \{P\} \cup \{P_1\})$, so from 5.24,

$$\beta \cup \{P\} \cup \{P_1\} \vdash D_2'' \rightsquigarrow Q. \tag{5.25}$$

Finally, 5.19 follows from 5.22 and 5.25, hence

$$\beta \cup \{P\} \vdash RR(D, \Phi \cup \{P\}) \rightsquigarrow Q.$$

This completes the induction. ■

Theorem 5.3 $D \rightarrowtail \mathfrak{U}(D)$.

Proof: We will prove that $D \rightarrowtail RR(D, \emptyset)$ by induction on D. When D is a claim or a primitive deduction, $RR(D, \emptyset) = D$, so the result is immediate since $\rightarrowtail$ is reflexive. When D is of the form **assume** P **in** D_b,

$$RR(D, \emptyset) = \textbf{assume } P \textbf{ in } RR(D_b, \{P\})$$

so in order to show $D \rightarrowtail RR(D, \emptyset)$ we need to prove that if

$$\beta \vdash \textbf{assume } P \textbf{ in } D_b \rightsquigarrow P \Rightarrow Q \tag{5.26}$$

then

$$\beta \vdash \textbf{assume } P \textbf{ in } RR(D_b, \{P\}) \rightsquigarrow P \Rightarrow Q. \tag{5.27}$$

On the assumption that 5.26 holds, we have

$$\beta \cup \{P\} \vdash D_b \rightsquigarrow Q. \tag{5.28}$$

By the inductive hypothesis, $D_b \rightarrowtail RR(D_b, \emptyset)$, so from 5.28 we get

$$\beta \cup \{P\} \vdash RR(D_b, \emptyset) \leadsto Q$$

and by Lemma 5.2, $\beta \cup \{P\} \vdash RR(D_b, \{P\}) \leadsto Q$. Therefore,

$$\beta \vdash \textbf{assume } P \textbf{ in } RR(D_b, \{P\}) \leadsto P \Rightarrow Q$$

which is the desired 5.27.

Finally, suppose that D is of the form $D_1; D_2$ and that $\beta \vdash D_1; D_2 \leadsto Q$, so that

$$\beta \vdash D_1 \leadsto P \tag{5.29}$$

and

$$\beta \cup \{P\} \vdash D_2 \leadsto Q. \tag{5.30}$$

We have $RR(D, \emptyset) = D_1'; D_2'$, where

$$D_1' = RR(D_1, \emptyset) \tag{5.31}$$

and

$$D_2' = RR(D_2, \mathcal{C}(D_1')). \tag{5.32}$$

From the inductive hypothesis, $D_1 \rightarrowtail RR(D_1, \emptyset)$, hence from 5.29,

$$\beta \vdash RR(D_1, \emptyset) \leadsto P \tag{5.33}$$

so from 5.31,

$$\beta \vdash D_1' \leadsto P \tag{5.34}$$

and

$$\mathcal{C}(D_1') = P. \tag{5.35}$$

Likewise, $D_2 \rightarrowtail RR(D_2, \emptyset)$, so from 5.30,

$$\beta \cup \{P\} \vdash RR(D_2, \emptyset) \leadsto Q$$

and from Lemma 5.2,

$$\beta \cup \{P\} \vdash RR(D_2, \{P\}) \leadsto Q$$

which, from 5.32 and 5.35 means

$$\beta \cup \{P\} \vdash D_2' \leadsto Q. \tag{5.36}$$

Finally, from 5.34 and 5.36 we infer that $\beta \vdash D_1'; D_2' \leadsto Q$, and from this we may conclude that $D \rightarrowtail RR(D, \emptyset) = D_1'; D_2'$. ■

5.2.3 Claim elimination

The third and final contracting transformation we will present is particularly simple: it eliminates all claims in non-trailing positions. It is readily verified that all such claims are superfluous. For example, the claim B in

$$D = \mathbf{dn}\ \neg\neg A; B; \mathbf{both}\ A, A$$

can be removed because $D \rightarrowtail \mathbf{dn}\ \neg\neg A; \mathbf{both}\ A, A$.

Claims in trailing positions cannot in general be removed, since they serve as conclusions. One exception, however, occurs when the claim P is the last element of a thread whose immediately preceding element concludes P. In those cases P can be removed despite its trailing position. An example is

$$\mathbf{dn}\ \neg\neg A; \mathbf{both}\ A, B; A \wedge B.$$

Here the trailing claim $A \wedge B$ can be weeded out because it is derived by the immediately dominating deduction $\mathbf{both}\ A, B$.

The following algorithm removes all claims in non-trailing positions, as well as all extraneous trailing claims of the sort discussed above:

$$\begin{array}{l}
\mathfrak{C}(D) = \\
\quad \mathit{match}\ \ D \\
\qquad \mathbf{assume}\ P\ \mathbf{in}\ D_b \to \mathbf{assume}\ P\ \mathbf{in}\ \mathfrak{C}(D_b) \\
\qquad D_1; D_2 \to\ \ \mathit{let}\ \ D_1' = \mathfrak{C}(D_1) \\
\qquad\qquad\qquad\qquad\ \ D_2' = \mathfrak{C}(D_2) \\
\qquad\qquad\qquad \mathit{in} \\
\qquad\qquad\qquad\quad \mathit{claim?}(D_1') \to D_2', \mathit{claim?}(D_2')\ \ \mathit{and}\ \ \mathcal{C}(D_1') = \mathcal{C}(D_2') \to D_1', D_1'; D_2'
\end{array}$$

where $\mathit{claim?}(D)$ returns true iff D is a claim. We have:

Lemma 5.4 (a) $P; D \rightarrowtail D$, *and* (b) $D; P \rightarrowtail D$ *whenever* $\mathcal{C}(D) = P$.

Using this lemma, a straightforward inductive argument will show that $D \rightarrowtail \mathfrak{C}(D)$. Termination is immediate.

Theorem 5.5 $\mathfrak{C}$ *always terminates. Further,* $D \rightarrowtail \mathfrak{C}(D)$.

5.3 Restructuring transformations

5.3.1 Scope maximization

The most fundamental restructuring transformation is *scope maximization*. Intuitively, this aims at making the conclusion of a subdeduction visible to as many subsequent parts of a deduction as possible. Scope can be limited in two ways: with bracketing (**begin-end** pairs), and with hypothetical deductions. We examine each case below.

Left-linear compositions

The first factor that can affect conclusion visibility is left-linear composition, namely, deductions of the form $(D_1; D_2); D_3$, where the conclusion of D_1 is only available to D_2. Such deductions are rare in practice because the natural threading style in $\mathcal{CND}$ is right-associative (which is why composition associates to the right by default). When they occur, left-linear compositions can complicate our parsimony analysis. Consider, for instance, $D = (D_1; D_2); D_3$ where $\mathcal{C}(D_1) = \mathcal{C}(D_3)$. Algorithm $\mathfrak{P}$ might well find D to be repetition-free even though, intuitively, it is clear that D_3 unnecessarily duplicates the work of D_1. The problem is the limited scope of D_1: as long as D_2 does not replicate the conclusion of D_1 and D_3 the conclusion of $D_1; D_2$, i.e., the conclusion of D_2, then D will be deemed repetition-free. The problem can be avoided by right-associating D, thereby maximizing the scope of D_1. The algorithm $\mathfrak{RL}$ that we present below converts every subdeduction of the form $(D_1; D_2); D_3$ into $D_1; (D_2; D_3)$. Our proof that this is a safe transformation will be based on Lemma 5.6 below. The proof of the lemma is straightforward and omitted, but the intuition is important: in both cases D_1 is available to D_2, and D_2 to D_3, but in $D_1; (D_2; D_3)$ we *also* have D_1 available to D_3. So if $(D_1; D_2); D_3$ goes through, then certainly $D_1; (D_2; D_3)$ will do so too.

Lemma 5.6 $(D_1; D_2); D_3 \rightarrowtail D_1; (D_2; D_3)$.

Specifically, let us say that a deduction D is *right-linear* iff $DLabel(D, u \oplus [1]) \neq ;$ for all $u \in Dom(D)$ such that $DLabel(D, u) = ;$. That is, D is right-linear iff it has no subdeductions of the form $(D_1; D_2); D_3$. The following is immediate:

Lemma 5.7 (a) *If D is right-linear then so is* **assume** P **in** D. (b) *If D_1 and D_2 are right-linear and D_1 is not a composite deduction, then $D_1; D_2$ is right-linear.*

Algorithm $\mathfrak{RL}$ will transform any given D into a right-linear D' such that $D \rightarrowtail D'$:

$$
\begin{array}{rcl}
\mathfrak{RL}(\textbf{assume } P \textbf{ in } D) & = & \textbf{assume } P \textbf{ in } \mathfrak{RL}(D) \\
 & & \textit{match } D_l \\
\mathfrak{RL}(D_l; D_r) & = & \quad D_1; D_2 \rightarrow \mathfrak{RL}(D_1; (D_2; D_r)) \\
 & & \quad \_ \rightarrow \mathfrak{RL}(D_l); \mathfrak{RL}(D_r) \\
\mathfrak{RL}(D) & = & D
\end{array}
$$

For our termination proof, let us write $SZ(D)$ to denote the size of D, and let us define a quantity $LSZ(D)$ as follows: if D is of the form $D_1; D_2$ then $LSZ(D) = SZ(D_1)$; otherwise $LSZ(D) = 0$. It immediately follows:

Lemma 5.8 (a) $LSZ((D_1; D_2); D_3) < LSZ(D_1; (D_2; D_3))$.

(b) $SZ((D_1; D_2); D_3) = SZ(D_1; (D_2; D_3))$.

Theorem 5.9 $\mathfrak{RL}$ *always terminates.*

Proof: We claim that with each recursive call, the pair $(SZ(D), LSZ(D))$ strictly decreases lexicographically.[3] This can be seen by checking each recursive call: in the recursive calls of the first two lines, the size of D strictly decreases. In the recursive call $\mathfrak{RL}(D_1; (D_2; D_r))$ the size does not increase (Lemma 5.8, part (b)), while the quantity LSZ strictly decreases ((Lemma 5.8, part (a)). Finally, in both recursive calls in the next line, the size strictly decreases. ■

Theorem 5.10 $\mathfrak{RL}(D)$ *is right-linear. Furthermore,* $D \rightarrowtail \mathfrak{RL}(D)$.

Proof: Let us write $D_1 \prec D_2$ to mean that the pair $(SZ(D_1), LSZ(D_1))$ is lexicographically smaller than $(SZ(D_2), LSZ(D_2))$. We will use well-founded induction on the relation $\prec$, i.e., we will show that for all deductions D, if the result holds for every D' such that $D' \prec D$ then it also holds for D. We proceed by a case analysis of an arbitrary D. If D is a claim or a primitive deduction then $\mathfrak{RL}(D) = D$ and the result follows immediately. If D is a hypothetical deduction with hypothesis P and body D' then $\mathfrak{RL}(D) =$ **assume** P **in** $\mathfrak{RL}(D')$. Since $D' \prec D$, the inductive hypothesis entails that $\mathfrak{RL}(D')$ is right-linear and that $D' \rightarrowtail \mathfrak{RL}(D')$. The result now follows from Lemma 5.7 and Lemma 4.22.

Finally, suppose that D is of the form $D_l; D_r$. Then either D_l is of the form $D_1; D_2$, or not. In the first case we have

$$\mathfrak{RL}(D) = \mathfrak{RL}(D_1; (D_2; D_r)) \tag{5.37}$$

and since $D = (D_1; D_2); D_r \succ D_1; (D_2; D_r)$, we conclude inductively that

(i) $\mathfrak{RL}(D_1; (D_2; D_r))$ is right linear, and

(ii) $D_1; (D_2; D_r) \rightarrowtail \mathfrak{RL}(D_1; (D_2; D_r))$.

Thus the conclusion that $\mathfrak{RL}(D)$ is right-linear follows from (i) and 5.37, while

$$D \rightarrowtail \mathfrak{RL}(D) = \mathfrak{RL}(D_1; (D_2; D_r))$$

[3]Using the lexicographic extension of $<$ to pairs of natural numbers: (n_1, n_2) is smaller than (n_1', n_2') iff $n_1 < n_1'$ or else $n_1 = n_1'$ and $n_2 < n_2'$.

follows from (ii) and the transitivity of $\rightarrowtail$, since $D \rightarrowtail D_1; (D_2; D_r)$ from Lemma 5.6. If D is not of the form $D_1; D_2$ then

$$\mathfrak{RL}(D) = \mathfrak{RL}(D_l); \mathfrak{RL}(D_r) \tag{5.38}$$

and since $D_l \prec D$, $D_r \prec D$, the inductive hypothesis entails that (i) $\mathfrak{RL}(D_l)$ and $\mathfrak{RL}(D_r)$ are right-linear, and (ii) $D_l \rightarrowtail \mathfrak{RL}(D_l)$, $D_r \rightarrowtail \mathfrak{RL}(D_r)$. Because D_l is not a composite deduction, neither is $\mathfrak{RL}(D_l)$ (a necessary condition for $\mathfrak{RL}(D)$ to be composite is that D be composite), hence it follows from part (b) of Lemma 5.7 and 5.38 that $\mathfrak{RL}(D)$ is right-linear. Further, $D \rightarrowtail \mathfrak{RL}(D)$ follows from (ii) and Lemma 4.22. This concludes the case analysis and the inductive argument. ■

Hypothetical deductions

The second case of undue scope limitation arises in hypothetical deductions. Consider a hypothetical deduction with body D_b and hypothesis P. If D is a subdeduction of D_b then its scope cannot extend beyond D_b. But this need not be the case if D is not strictly dependent on the hypothesis P. If there is no such dependence, then D is unnecessarily restricted by being inside D_b. Its scope should be maximized by *hoisting* it outside D_b. As a simple example, consider

assume B **in**
  begin
  double-negation $\neg\neg A$;
  both A, B
  end

Here the subdeduction **double-negation** $\neg\neg A$ makes no use of the hypothesis B, and therefore it is appropriate to pull it outside, resulting in

double-negation $\neg\neg A$;
assume B **in**
  both A, B

This deduction is observationally equivalent to the first one, and has a cleaner structure that better reflects the various logical dependencies. Besides increased clarity, hoisting will greatly facilitate our repetition analysis later on. Repetitions are much easier to detect and eliminate when they are in the same scope. Consider, for instance, the deduction

assume B **in**
  begin
    double-negation $\neg\neg A$;
    both A, B
  end;
left-and $A \wedge C$;
both $A, B \Rightarrow A \wedge B$

In view of **double-negation** $\neg\neg A$, the deduction **left-and** $A \wedge C$ is superfluous, but this is not easy to determine mechanically because the former deduction lies inside the scope of the hypothesis B. More importantly, neither deduction can be safely eliminated as things stand, even though it is clearly extraneous to have both of them. If we eliminated the **double-negation** then the **assume** might fail; while if we eliminated the **left-and**, the composition might fail. But if we hoist the double negation outside of the **assume**, resulting in

double-negation $\neg\neg A$;
assume B **in**
  both A, B;
left-and $A \wedge C$;
both $A, B \Rightarrow A \wedge B$

then the repetition becomes much easier to detect, and the **left-and** can be confidently eliminated.

In what follows we will be dealing with lists of deductions $[D_1, \ldots, D_n]$. We will use the letter Δ for such lists, and, as usual, the symbol $\oplus$ for list concatenation. For a non-empty list $\Delta = [D_1, \ldots, D_n]$, $n > 0$, we define $\overline{\Delta}$ as the thread $D_1; \ldots; D_n$. The following will come handy later:

Lemma 5.11 $\overline{\Delta_1 \oplus \Delta_2} = \overline{\Delta_1}; \overline{\Delta_2}$

We adopt the convention that when Δ is empty the expresssion $\overline{\Delta}; D$ stands for D.

The algorithm H in Figure 5.3 examines a right-linear thread $D = D_1; \ldots; D_n$ (we make the simplifying convention that we might have $n = 1$, in which case D_1 will not be composite, since we are assuming that D is right-linear) and pulls out every D_i that is not transitively dependent on a set of assumptions Φ. Each hoisted D_i is replaced in-place in D by the claim $\mathcal{C}(D_i)$. Specifically, $H(D, \Phi)$ returns a triple (D', Ψ, Δ), where

- D' is obtained from D by replacing every D_i that does not transitively depend on Φ by $\mathcal{C}(D_i)$.

$$
\begin{array}{rcl}
H(D_1; D_2, \Phi) & = & \begin{array}{l} \mathit{let}\ (D_1', \Phi_1, \Delta_1) = H(D_1, \Phi) \\ \qquad (D_2', \Phi_2, \Delta_2) = H(D_2, \Phi_1) \\ \mathit{in} \\ \quad (D_1'; D_2', \Phi_2, \Delta_1 \oplus \Delta_2) \end{array} \\
H(D, \Phi) & = & \mathit{FA}(D) \cap \Phi = \emptyset \rightarrow (\mathcal{C}(D), \Phi, [D]), (D, \Phi \cup \{\mathcal{C}(D)\}, [])
\end{array}
$$

Figure 5.3: The kernel of the hoisting algorithm.

- $\Psi \supseteq \Phi$ is monotonically obtained from Φ by incorporating the conclusions of those deductions D_j that do depend (transitively) on Φ. This is essential in order to handle transitive dependence.
- Δ is a list $[D_{i_1}, \ldots, D_{i_k}]$, $1 \leq i_j \leq n$, $j = 1, \ldots, k \geq 0$, of those deductions that do not depend on Φ. The order is important for preserving dominance constraints: we have $i_a < i_b$ for $a < b$, since, e.g., D_5 and D_8 might not be dependent on Φ, but D_8 might depend on D_5. Accordingly, Δ should respect the original ordering.

As Theorem 5.15 will prove, the idea is that we will have $D \rightarrowtail \overline{\Delta}; D'$. The thread $D_1; \cdots; D_n$ should be thought of as the body of a hypothetical deduction with hypothesis P, and Φ should be thought of as $\{P\}$. Then if $H(D_1; \ldots; D_n, \Phi) = (D', \Psi, \Delta)$, D' will be the new body of the hypothetical deduction, and the thread $\overline{\Delta}$ will comprise the hoisted deductions, with a dominance relation that respects the original ordering $1, \ldots, n$.

Lemma 5.12 *Let* $(D_2, \Phi_2, \Delta) = H(D_1, \Phi_1)$. *Then* $\Phi_1 \cap \mathit{FA}(D) = \emptyset$ *for every* $D \in \Delta$.

Proof: By induction on D_1. Suppose first that D_1 is not composite. There are two cases: either $\mathit{FA}(D_1) \cap \Phi_1 = \emptyset$ or not. If not then $\Delta = []$ so the result holds vacuously. If $\mathit{FA}(D_1) \cap \Phi_1 = \emptyset$ then $\Delta = U(D_1)$ so the result holds by supposition. Finally, if D_1 is of the form $D_l; D_r$ then $\Delta = \Delta_l \oplus \Delta_r$, where $H(D_l, \Phi_1) = (D_l', \Phi_l, \Delta_l)$ and $H(D_r, \Phi_l) = (D_r', \Phi_r, \Delta_r)$. Inductively,

$$\forall D \in \Delta_l, \Phi_1 \cap \mathit{FA}(D) = \emptyset \tag{5.39}$$

and

$$\forall D \in \Delta_r, \Phi_l \cap \mathit{FA}(D) = \emptyset \tag{5.40}$$

Since $\Phi_1 \subseteq \Phi_l$, 5.40 entails

$$\forall D \in \Delta_r, \Phi_1 \cap \mathit{FA}(D) = \emptyset \tag{5.41}$$

The result now follows from 5.39 and 5.41 since $\Delta = \Delta_l \oplus \Delta_r$, and the inductive argument is thus complete. ∎

We will also need the following two results, whose proofs are simple and omitted:

Lemma 5.13 *If $\mathcal{C}(D) \notin FA(D_i)$ for $i = 1, \ldots, n$ then*

$$D; D_1; \ldots; D_n; D' \rightarrowtail D_1; \ldots; D_n; D; D'.$$

Lemma 5.14 *$P; D_1; \ldots; D_n; D \rightarrowtail D_1; \ldots; D_n; P; D$.*

Theorem 5.15 *If D is right-linear and $(D', \Psi, \Delta) = H(D, \Phi)$ then $D \rightarrowtail \overline{\Delta}; D'$.*

Proof: By induction on D. Suppose first that D is not a composition. Then either $FA(D) \cap \Phi = \emptyset$ or not. If not, then $D' = D$ and $\Delta = []$, so the result is immediate. If $FA(D) \cap \Phi = \emptyset$ then $D' = \mathcal{C}(D)$ and $\Delta = [D]$, so again the result follows directly.

Suppose next that D is of the form $D_1; D_2$. Then, letting

$$(D'_1, \Phi_1, \Delta_1) = H(D_1, \Phi) \tag{5.42}$$

and

$$(D'_2, \Phi_2, \Delta_2) = H(D_2, \Phi_1) \tag{5.43}$$

we have $D' = D'_1; D'_2$ and $\Delta = \Delta_1 \oplus \Delta_2$, so we have to show

$$D \rightarrowtail \overline{\Delta_1 \oplus \Delta_2}; D'_1; D'_2. \tag{5.44}$$

From 5.42, 5.43, and the inductive hypothesis, we have

$$D_1 \rightarrowtail \overline{\Delta_1}; D'_1 \tag{5.45}$$

and

$$D_2 \rightarrowtail \overline{\Delta_2}; D'_2 \tag{5.46}$$

Therefore,

$$D = D_1; D_2 \rightarrowtail \overline{\Delta_1}; D'_1; \overline{\Delta_2}; D'_2 \tag{5.47}$$

Now since we are assuming that D is right-linear, D_1 cannot be composite, hence again we distinguish two cases: $FA(D_1) \cap \Phi = \emptyset$, or not. If the latter holds then $D'_1 = D_1, \Phi_1 = \Phi \cup \{\mathcal{C}(D_1)\}$, and $\Delta_1 = []$. Now by 5.43 and Lemma 5.12 we have that, for every $D_x \in \Delta_2$, $\Phi_1 \cap FA(D_x) = \emptyset$, and since $\mathcal{C}(D_1) \in \Phi_1$, this means that $\mathcal{C}(D_1) \notin FA(D_x)$. Hence, from Lemma 5.13,

$$D'_1; \overline{\Delta_2}; D'_2 \rightarrowtail \overline{\Delta_2}; D'_1; D'_2$$

and thus

$$\overline{\Delta_1}; D_1'; \overline{\Delta_2}; D_2' \rightarrowtail \overline{\Delta_1}; \overline{\Delta_2}; D_1'; D_2'. \tag{5.48}$$

On the other hand, if $FA(D_1) \cap \Phi = \emptyset$ then $D_1' = \mathcal{C}(D_1)$, so by Lemma 5.14 we have

$$D_1'; \overline{\Delta_2}; D_2' \rightarrowtail \overline{\Delta_2}; D_1'; D_2'$$

and hence 5.48 follows again. Thus we have shown that in either case 5.48 holds, and since $\overline{\Delta_1 \oplus \Delta_2} = \overline{\Delta_1} \oplus \overline{\Delta_2}$, it now follows from 5.47, 5.48, and the transitivity of $\rightarrowtail$ that

$$D \rightarrowtail \overline{\Delta_1 \oplus \Delta_2}; D_1'; D_2'$$

which is 5.44, exactly what we wanted to show. This completes the induction. ■

As an illustration of the algorithm, let D be the deduction

1. **modus-ponens** $A \Rightarrow C \wedge B, A$;
2. **double-negation** $\neg\neg B$;
3. **left-and** $C \wedge B$;
4. **right-either** C, B;
5. **both** C, C

and consider the call $H(D, \{A\})$. Let D_a–D_e refer to the deductions in lines 1–5, respectively. Since D is composite, the first clause of the algorithm will be chosen, so the first recursive call will be $H(D_a, \{A\})$, which, since D_a is not composite and $FA(D_a) \cap \{A\} \neq \emptyset$, will yield the result $(D_a, \{A, C \wedge B\}, [])$. The second recursive call is $H(D_b; D_c; D_d; D_e, \{A, C \wedge B\})$. This in turn gives rise to the recursive calls $H(D_b, \{A, C \wedge B\})$, which returns

$$(B, \{A, C \wedge B\}, [\textbf{double-negation } \neg\neg B])$$

and $H(D_c; D_d; D_e, \{A, C \wedge B\})$. The latter will spawn $H(D_c, \{A, C \wedge B\})$, which will return $(D_c, \{A, C \wedge B\}, [])$, and $H(D_d; D_e, \{A, C \wedge B, C\})$. In the same fashion, the latter will spawn $H(D_d, \{A, C \wedge B, C\})$, which will return

$$(D_d, \{A, C \wedge B, C\}, [\textbf{right-either } C, B])$$

and $H(D_e, \{A, C \wedge B, C\})$, which will produce $(D_e, \{A, C \wedge B, C, C \wedge C\}, [])$. Moving up the recursion tree will eventually yield the final result (D', Ψ, Δ), where D' is the deduction

1.**modus-ponens** $A \Rightarrow C \wedge B, A$;
2.B;
3.**left-and** $C \wedge B$;
4.$C \vee B$;
5.**both** C, C

Ψ is the set $\{A, C \wedge B, C, C \wedge C\}$, and Δ is the list

$$[\textbf{double-negation } \neg\neg B, \textbf{right-either } C, B].$$

Thus $\overline{\Delta}; D'$ is the deduction

double-negation $\neg\neg B$;
right-either C, B

---

modus-ponens $A \Rightarrow C \wedge B, A$;
B;
left-and $C \wedge B$;
$C \vee B$;
both C, C

The horizontal line demarcates the hoisted deductions from D'.

If D were the body of a hypothetical deduction with hypothesis A, then the result of the hoisting would be

$$\overline{\Delta}; \textbf{assume } A \textbf{ in } D'$$

namely,

double-negation $\neg\neg B$;
right-either C, B;
assume A **in**
  begin
    modus-ponens $A \Rightarrow C \wedge B, A$;
    B;
    left-and $C \wedge B$;
    $C \vee B$;
    both C, C
  end

A subsequent contracting transformation to remove claims (algorithm $\mathfrak{C}$) would result in

double-negation $\neg\neg B$;
right-either C, B;
assume A **in**
  begin
    modus-ponens $A \Rightarrow C \wedge B, A$;
    left-and $C \wedge B$;
    both C, C
  end

The hoisting transformation should be applied to every hypothetical deduction contained in a given D. This must be done in stages and in a bottom-up direction in order for hoisted inferences to "bubble" as far up as possible (to maximize their scope). Specifically, let D be a given deduction. The hoisting will proceed in stages $i = 1, \ldots, n, \ldots$, where we begin with $D_1 = D$. At each stage i we replace certain *candidate* hypothetical subdeductions of D_i by new deductions, and the result we obtain from these replacements becomes D_{i+1}. We keep going until we reach a fixed point, i.e., until $D_{i+1} = D_i$.

At each point in the process every hypothetical subdeduction of D_i is either *marked*, indicating that its body has already been processed, or unmarked. An invariant we will maintain throughout is that a marked hypothetical subdeduction will never contain unmarked hypothetical deductions; this will be enforced by the way in which we will be choosing our candidates, and will ensure that hoisting will proceed in a bottom-up direction. Initially, all hypothetical subdeductions of $D_1 = D$ are unmarked. On stage i, an unmarked hypothetical subdeduction of D_i is a candidate for hoisting iff it is as deep as possible, i.e., iff it does not itself contain any unmarked hypothetical subdeductions. For each such candidate $D_c =$ **assume** P **in** D_b occuring in position $u \in Dom(D_i)$, we compute $(D'_b, \Psi, \Delta) = H(D_b, \{P\})$, and we replace D_c in position u of D_i by $\overline{\Delta}$; $\underline{\textbf{assume}}$ P **in** D'_b, where the **assume** is now marked to indicate that its body D'_b has been combed bottom-up and we are thus finished with it—it can no longer serve as a candidate. Note that because all candidate subdeductions of D_i are pairwise disjoint (i.e., none of them is contained in another candidate), these replacements could in principle occur in parallel. The deduction we obtain from D_i by carrying out these replacements becomes D_{i+1}. One pitfall to be avoided: the replacements might introduce left-linear subdeductions into D_{i+1}. Algorithm H, however, expects its argument to be right-linear, so after the replacements are performed we need to apply $\mathfrak{RL}$ to D_{i+1} before continuing with the next stage.

Algorithm *Hoist* below replaces every candidate hypothetical subdeduction of a given D in the manner discussed above and marks the processed subdeduction:

$$
\begin{array}{l}
\mathit{Hoist}(D) = \mathit{match}\ \ D \\
\qquad\qquad\qquad \textbf{assume}\ P\ \textbf{in}\ D_b \rightarrow \\
\qquad\qquad\qquad\quad \mathit{Is\ every}\ \textbf{assume}\ \mathit{within}\ D_b\ \mathit{marked?} \rightarrow \\
\qquad\qquad\qquad\qquad \mathit{let}\ \ (D'_b, \_\,, \Delta) = H(D_b, \{P\}) \\
\qquad\qquad\qquad\qquad \mathit{in} \\
\qquad\qquad\qquad\qquad\quad \overline{\Delta}; \underline{\textbf{assume}}\ P\ \textbf{in}\ D'_b, \\
\qquad\qquad\qquad\qquad \textbf{assume}\ P\ \textbf{in}\ \mathit{Hoist}(D_b) \\
\qquad\qquad\qquad D_1; D_2 \rightarrow \mathit{Hoist}(D_1); \mathit{Hoist}(D_2) \\
\qquad\qquad\qquad D \rightarrow D
\end{array}
$$

We can now formulate our final scope-maximization transformation as follows:

$$
\begin{array}{l}
\mathfrak{MS}(D) = \mathit{FP}(\mathfrak{RL}(D)) \\
\text{where} \\
\mathit{FP}(D) = \mathit{let}\ \ D' = \mathfrak{RL}(\mathit{Hoist}(D)) \\
\qquad\qquad\ \mathit{in} \\
\qquad\qquad\quad D' = D \rightarrow D, \mathit{FP}(D')
\end{array}
$$

That $\mathfrak{MS}$ always terminates follows from the fact that *Hoist* does not introduce any additional hypothetical deductions, and either outputs the same result unchanged (a fixed point) or a deduction with at least one more hypothetical deduction marked. Since any deduction only has a finite number of hypothetical subdeductions, this means that $\mathfrak{MS}$ will eventually converge to a fixed point. Further, $D \rightarrowtail \mathfrak{MS}(D)$ follows from the corresponding property of $\mathfrak{RL}$, from Theorem 5.15, and from the transitivity and compatibility of the $\rightarrowtail$ relation. Finally, Lemma 5.12 gives:

Theorem 5.16 (a) $\mathfrak{MS}$ *always terminates;*

(b) $\mathfrak{MS}(D)$ *is right-linear;*

(c) $D \rightarrowtail \mathfrak{MS}(D)$*;*

(d) *the proof-composition graph of the body of every hypothetical deduction of* $\mathfrak{MS}(D)$ *is proper, i.e., connected, pointed, and acyclic.*

5.3.2 Global transformations of hypothetical deductions

The hoisting algorithm is a focused, local transformation: we delve inside a given deduction D and work on subdeductions of the form **assume** P **in** D_b, taking into account only the hypothesis P and the body D_b. We do not utilize any knowledge from a wider context. More intelligent tranformations become possible if we look at the big picture, namely, at how P and D_b relate to other parts of the enclosing deduction D. In this section we will present three such transformations, $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$. All three of them perform a global analysis of a given deduction D and replace every hypothetical subdeduction D' of it by some other deduction D'' (where we might have $D'' = D'$). These transformations expect their input deductions to have been processed by $\mathfrak{MS}$, but their output deductions might contain left-linear compositions or hoisting possibilities that were not previously visible. It is for this reason that their composition must be interleaved with the scope-maximization procedure $\mathfrak{MS}$, as specified in 5.3.

The first transformation, $\mathfrak{A}_1$, targets every hypothetical subdeduction of D of the form $D' = \textbf{assume } P \textbf{ in } D_b$ whose hypothesis P is a free assumption of D, i.e., such that $P \in FA(D)$. Clearly, D can only be successfully evaluated in an assumption base that contains P (Corollary 4.11). But if we must evaluate D in an assumption base that contains P then there is no need to postulate P hypothetically and hide D_b behind that hypothesis; we can pull D_b outside. Accordingly, this analysis will replace D' by the thread $D'' = D_b; \textbf{assume } P \textbf{ in } \mathcal{C}(D_b)$. Thus the final conclusion is unaffected (it is still the conditional $P \Rightarrow \mathcal{C}(D_b)$), but the scope of D_b is enlarged. Specifically, we define:

$$
\begin{aligned}
&\mathfrak{A}_1(D) = T(D, FA(D))\\
&\text{where}\\
&T(\textbf{assume } P \textbf{ in } D_b, \Phi) = \mathit{let}\ D_b' = T(D_b, \Phi)\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \mathit{in}\\
&\qquad\qquad\qquad\qquad\qquad\qquad P \in \Phi \to D_b'; \textbf{assume } P \textbf{ in } \mathcal{C}(D_b'), \textbf{assume } P \textbf{ in } D_b'\\
&T(D_l; D_r, \Phi) = T(D_l, \Phi); T(D_r, \Phi)\\
&T(D, \Phi) = D
\end{aligned}
$$

Note that we first process D_b recursively and then pull it out, since D_b might itself contain hypothetical deductions with a free assumption as a hypothesis. As an example, if D is the deduction

assume A **in**
 begin
  both A, A;
  assume B **in**
   both $B, A \wedge A$
 end;
both A, B

where both conditional deductions have free assumptions as hypotheses (A and B) then $\mathfrak{A}_1(D)$ will be

both A, A;
both $B, A \wedge A$;
assume B **in**
  $B \wedge A \wedge A$;
assume A **in**
  $B \Rightarrow B \wedge A \wedge A$;
both A, B

The two remaining transformations turn not on whether the hypothesis of a conditional deduction is a free assumption, but on whether it is deduced at some prior or subsequent point. For the second transformation, $\mathfrak{A}_2$, suppose that during our evaluation of D we come to a conditional subdeduction $D' = \textbf{assume } P \textbf{ in } D_b$ whose hypothesis P either has already been established or else has already been hypothetically postulated (i.e., D' is itself nested within an **assume** with hypothesis P). Then we may again pull D_b out, replacing D' by the composition $D'' = D_b; \textbf{assume } P \textbf{ in } \mathcal{C}(D_b)$. (More precisely, just as in $\mathfrak{A}_1$, we first have to process D_b recursively before hoisting it.) Thus we arrive at the following algorithm:

$\mathfrak{A}_2(D) = T(D, \emptyset)$
where
$T(D, \Phi) = \mathit{match}\ D$
    $\textbf{assume } P \textbf{ in } D_b \rightarrow$
      $P \in \Phi \rightarrow \ \mathit{let}\ \ D'_b = T(D_b, \Phi)$
        in
          $D'_b; \textbf{assume } P \textbf{ in } \mathcal{C}(D'_b)$,
        $\textbf{assume } P \textbf{ in } T(D_b, \Phi \cup \{P\})$
    $D_1; D_2 \rightarrow \ \mathit{let}\ \ D'_1 = T(D_1, \Phi)$
      in
        $D'_1; T(D_2, \Phi \cup \{\mathcal{C}(D_1)\})$
    $D \rightarrow D$

As a simple example, if D is the deduction

dn $\neg\neg A$;
assume A **in**
  both A, B;
both $A \wedge B, A \Rightarrow A \wedge B$

then

$$\mathfrak{A}_2(D) = \begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{both } A, B; \\ \textbf{assume } A \textbf{ in} \\ \quad A \wedge B; \\ \textbf{both } A \wedge B, A \Rightarrow A \wedge B \end{array}$$

The reader will notice a similarity in the structure of algorithms $\mathfrak{P}$ and $\mathfrak{A}_2$. Both of them proceed in a top-down manner, cumulatively recording all intermediate conclusions established along a thread, as well as hypotheses postulated by nested conditional deductions. In fact in a practical implementation of our optimization procedure the two algorithms should be combined and performed in lock-step in the interest of efficiency. There are other transformations that could also be combined and other modifications that could be made that would result in a more efficient implementation of *normalize* than the one suggested by definitions 5.1, 5.2 and 5.3. These particular definitions were given for their conceptual benefits—especially because they separate the contracting from the restructuring transformations—in order to help the reader to better understand the overall process. A practical implementation of *normalize* should achieve the extensional behavior specified by the above equations, but in a more efficient manner. Finally, the third transformation, $\mathfrak{A}_3$, determines whether the hypothesis P of a conditional deduction $D' = \textbf{assume } P \textbf{ in } D_b$ is deduced at a later point, or, more precisely, whether it is deduced somewhere within a deduction dominated by D', as in the following picture:

$$\begin{array}{ll} \vdots & \\ D' = \textbf{assume } P \textbf{ in } D_b; & \\ \vdots & \\ D''; & \text{(Deduces } P\text{)} \\ \vdots & \end{array}$$

To motivate this transformation, consider the following deduction:

(1) **assume** $\neg\neg A$ **in**
    begin
      dn $\neg\neg A$;
      both A, B
    end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **modus-ponens** $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$

This deduction illustrates one of the detours we discussed earlier, whereby Q is derived by first inferring $P \Rightarrow Q$, then P, and then using **modus-ponens** on $P \Rightarrow Q$

and P. The detour arises because the hypothesis P is in fact deducible, and hence there is no need for the implication $P \Rightarrow Q$ and the **modus-ponens**. We can simply deduce P and then directly perform the reasoning of the body of the hypothetical deduction.

We note that unlike the cases discussed in connection with $\mathfrak{A}_2$ and $\mathfrak{A}_1$, here we cannot hoist the body of (1) above the **assume** (and insert in its place the claim $A \wedge B$), because the said body strictly uses the hypothesis $\neg\neg A$, which is neither a free assumption of the overall deduction (which would be the province of $\mathfrak{A}_1$) nor is it deduced *prior* to its hypothetical postulation in (1) (which would be the province of $\mathfrak{A}_2$). Rather, $\neg\neg A$ is deduced *after* the conditional deduction where it appears as a hypothesis. So what $\mathfrak{A}_3$ will do is the following: it will insert a *copy* of the body of the conditional deduction (1) immediately after the hypothesis $\neg\neg A$ is subsequently deduced, i.e., immediately after (2), resulting in:

(1) **assume** $\neg\neg A$ **in**
    begin
      dn $\neg\neg A$;
      both A, B
    end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **dn** $\neg\neg A$;
(4) **both** A, B;
(5) **modus-ponens** $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$

(in fact we have to insert the body of the hypothetical deduction as is, including the **begin-end** brackets, which will result in a deduction that is not right-linear. However, a subsequent pass of $\mathfrak{MS}$ will make the deduction right-linear and the **begin-end** pair will be eliminated, resulting in the above; see the remark on page 128 in reference to the interleaving of $\mathfrak{MS}$). This increases the size of the deduction, but as we will see shortly the increase is temporary. First the reader should observe that this is a safe transformation, because, by virtue of the monotonic evaluation of threads, everything that the body of (1) might require in the way of free assumptions is also available at the point where the copy is inserted, *including the proposition* $\neg\neg A$.

Now after the transformation has taken place, our parsimony analysis $\mathfrak{P}$ will detect that (5) derives the same conclusion as the preceding (4), and will thus replace it by the claim $A \wedge B$. A subsequent utility analysis will reveal that the conclusion of (1) is no longer used, and will thus eliminate (1). Finally, our claim-removal procedure $\mathfrak{C}$ will also remove the trailing claim $A \wedge B$ (it is extraneous, since it is immediately preceded by a deduction which infers it), resulting in the detour-free

left-and $\neg\neg A \wedge C$;

dn $\neg\neg A$;
both A, B

In general, $\mathfrak{A}_3$ determines whether a hypothesis P of a subdeduction

$$D' = \textbf{assume } P \textbf{ in } D_b$$

is derived by a deduction D'' rooted at a position u that is dominated by D'. If so, it must then decide whether to replace D'' by the composition $D''; D_b$. Because such a replacement will increase the size of the overall deduction, we must take care to avoid it unless we can be certain that the preceding **assume** will be subsequently eliminated by the contracting analyses, as in the above example. Clearly, one case where the copying is not necessary occurs when P is not used as an argument to an offending **modus-ponens**. In such a case there is no detour to speak of, since a detour exists only if the derivation of P is followed up by a **modus-ponens** detachment of $\mathcal{C}(D_b)$. As an example, consider the following variation of the previous example:

(1) **assume** $\neg\neg A$ **in**
  begin
   dn $\neg\neg A$;
   both A, B
  end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **both** $\neg\neg A, \neg\neg A \Rightarrow A \wedge B$

Here there is no detour, and hence no need for inserting the body of (1) right after the hypothesis $\neg\neg A$ is derived at point (2). Nevertheless, it is interesting to note that even if we did perform the copying, resulting in

(1) **assume** $\neg\neg A$ **in**
  begin
   dn $\neg\neg A$;
   both A, B
  end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **dn** $\neg\neg A$;
(4) **both** A, B;
(5) **both** $\neg\neg A, \neg\neg A \Rightarrow A \wedge B$

the subsequent utility $\mathfrak{U}$ transformation would eliminate both (4) and (3), since their conclusions are not needed by (5), and would thus finally reproduce the original

assume $\neg\neg A$ **in**
  begin
    dn $\neg\neg A$;
    both A, B
  end;
left-and $\neg\neg A \wedge C$;
both $\neg\neg A, \neg\neg A \Rightarrow A \wedge B$

However, this is wasted effort, and we prefer to simply avoid the copying in the first place if we conclude that there is no detour.

But now observe that there might be no detour *even* if the derivation of P is followed by a **modus-ponens** detachment of $\mathcal{C}(D_b)$. This will be the case if the conclusion of the **assume**, namely $P \Rightarrow \mathcal{C}(D_b)$, has additional uses over and above its role in the detachment of $\mathcal{C}(D_b)$. As an example, consider the deduction

$$D = \begin{array}{l} (1)\ \textbf{assume}\ \neg\neg A\ \textbf{in} \\ \qquad \textbf{begin} \\ \qquad\quad \textbf{dn}\ \neg\neg A; \\ \qquad\quad \textbf{both}\ A, B \\ \qquad \textbf{end}; \\ (2)\ \textbf{left-and}\ \neg\neg A \wedge C; \\ (3)\ \textbf{modus-ponens}\ \neg\neg A \Rightarrow A \wedge B, \neg\neg A; \\ (4)\ \textbf{both}\ A \wedge B, \neg\neg A \Rightarrow A \wedge B \end{array} \tag{5.49}$$

Here we do have the requisite **modus-ponens** in (3), following the derivation of $\neg\neg A$ in (2), and this would appear to indicate a detour. However, there is actually no detour because the conclusion of the hypothetical deduction (1) is *also* used in (4) as an indispensable part of the final conclusion. This means that we could not possibly eliminate (1); it is essential, not a detour. Indeed, here is what we would obtain if we went ahead and performed the copying:

(1) **assume** $\neg\neg A$ **in**
    begin
      dn $\neg\neg A$;
      both A, B
    end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **dn** $\neg\neg A$;
(4) **both** A, B;

(5) **modus-ponens** $\neg\neg A \Rightarrow A \wedge B, \neg\neg A$;
(6) **both** $A \wedge B, \neg\neg A \Rightarrow A \wedge B$

Now the parsimony analysis would admittedly eliminate (5), resulting in

(1) **assume** $\neg\neg A$ **in**
  begin
   dn $\neg\neg A$;
   both A, B
  end;
(2) **left-and** $\neg\neg A \wedge C$;
(3) **dn** $\neg\neg A$;
(4) **both** A, B;
(5) **both** $A \wedge B, \neg\neg A \Rightarrow A \wedge B$

But now note that the utility analysis would not be able to eliminate (1), and rightly so, precisely because its conclusion is used in the final inference (5). Thus we would be left with a net size increase.

Of course this is not to say that deduction 5.49 is optimal. The proper way to write it is

left-and $\neg\neg A \wedge C$;
dn $\neg\neg A$;
both A, B;
assume $\neg\neg A$ **in**
  $A \wedge B$
both $A \wedge B, \neg\neg A \Rightarrow A \wedge B$

which is slightly more efficient (it avoids the **modus-ponens**). However, because the conclusion of the **assume** is essential, 5.49 does not have the same kind of detour as the deduction in page 130.

In general, consider a thread of the form

$$\begin{array}{l}\textbf{assume } P \textbf{ in } D_b; \\ D\end{array} \tag{5.50}$$

(where D might itself be a thread), and let $\mathcal{C}(D_b) = Q$. We will say that the hypothetical deduction **assume** P **in** D_b is *a detour in* D iff every strict use of $P \Rightarrow Q$ in D (with the exception of certain trivial claims)

(a) occurs in a primitive deduction of the form **modus-ponens** $P \Rightarrow Q, P$; and

(b) is dominated by a deduction D_P with $\mathcal{C}(D_P) = P$.

The following function takes a deduction D and a conditional $P \Rightarrow Q$ and determines whether the above two conditions hold:

$$
\begin{array}{l}
\textit{detour?}(D, P \Rightarrow Q) = g(D, 0) \\
\text{where} \\
g(D, b) = \textit{match}\ \ D \\
\qquad\qquad R \rightarrow [R \neq (P \Rightarrow Q) \rightarrow \textit{true}, \textit{false}] \\
\qquad\qquad \textit{Prim-Rule}\ \ P_1, \ldots, P_n \rightarrow [(P \Rightarrow Q) \notin \textit{FA}(D) \rightarrow \textit{true}, h(D, b)] \\
\qquad\qquad \textbf{assume}\ P_1\ \textbf{in}\ D_1 \rightarrow [P_1 = (P \Rightarrow Q) \rightarrow \textit{true}, g(D_1, b)] \\
\qquad\qquad D_1; D_2 \rightarrow [g(D_1, b) \rightarrow (\mathcal{C}(D_1) = P \rightarrow g(D_2, 1), g(D_2, b)), \\
\qquad\qquad\qquad\qquad\qquad (\textit{claim?}(D_1) \rightarrow g(D_2, b), \textit{false})]
\end{array}
$$

where $h(D, b)$ returns *true* if D is of the form **modus-ponens** $P \Rightarrow Q, P$ *and* $b = 1$, and *false* otherwise. The bit b is used to keep track of whether we are within the scope of a deduction D_P with conclusion P, as required by (b). Also note that in the case of **assume** P_1 **in** D_1, if $P_1 = (P \Rightarrow Q)$ then, by the preceding analysis ($\mathfrak{A}_2$), the body D_1 must be a single claim, since we are assuming that D is dominated by a hypothetical deduction with conclusion $P \Rightarrow Q$ (see 5.50).

If *detour?* yields a positive verdict then we may go ahead with the copying, confident that the corresponding hypothetical deduction will ultimately be removed by the contracting transformations. Thus we can now express $\mathfrak{A}_3$ as shown below. One final complicating factor is that one and the same proposition P might be the hypothesis of several different conditional subdeductions, so when P is later deduced we need to insert local copies of the body of every conditional deduction that has P as its hypothesis, provided of course that these hypothetical deductions are indeed detours (as determined by *detour?*). Specifically:

$$
\begin{array}{l}
\mathfrak{A}_3 = T(D, []) \\
\text{where} \\
T(D, L) = \\
\quad \textit{match}\ \ D \\
\qquad D_1; D_2 \ \rightarrow\ \ \textit{let}\ \ (\Delta, M) = \textit{find}(\mathcal{C}(D_1), L) \\
\qquad\qquad\qquad\quad \textit{in} \\
\qquad\qquad\qquad\qquad \textit{match}\ D_1 \\
\qquad\qquad\qquad\qquad\quad \textbf{assume}\ P\ \textbf{in}\ D_b \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \textit{let}\ \ M' = \textit{detour?}(D_2, P \Rightarrow \mathcal{C}(D_b))\ \textit{and}\ \textit{NT}(D_b) \rightarrow \langle P, D_b \rangle{::}M, M \\
\qquad\qquad\qquad\qquad\qquad \textit{in}
\end{array}
$$

$$(\textbf{assume } P \textbf{ in } T(D_b, L)); \overline{\Delta}; T(D_2, M')$$
$$D' \to D'; \overline{\Delta}; T(D_2, M)$$

$D' \to D'$

and

$\mathit{find}(P, L) = f(L, [], L)$

where

$f([], \Delta, M) = (\Delta, M)$

$f(\langle Q, D\rangle{::}L', \Delta, M) = P = Q \to f(L', D{::}\Delta, \mathit{remove}(\langle Q, D\rangle, M)), f(L', \Delta, M)$

while

- $\mathit{remove}(x, L)$ removes all copies of an element x from a list L;
- $NT(D) = \mathit{not}(\mathit{claim?}(D))$,i.e., *true* if D is a non-trivial deduction and *false* otherwise.

A more efficient implementation could be obtained by combining *detour?* and $\mathfrak{A}_3$.

5.4 Examples

In this section we illustrate *normalize* with some simple examples of the "detours" shown in Figure 5.2. We begin with a couple of examples of detour (c):

$$\mathrm{D} = \begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{assume } A \textbf{ in} \\ \quad \textbf{both } A, B; \\ \textbf{mp } A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\mathit{restructure}} \begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{both } A, B; \\ \textbf{assume } A \textbf{ in} \\ \quad A \wedge B; \\ \textbf{mp } A \Rightarrow A \wedge B, A \end{array} \xrightarrow{\mathfrak{P}}$$

$$\begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{both } A, B; \\ \textbf{assume } A \textbf{ in} \\ \quad A \wedge B; \\ A \wedge B \end{array} \xrightarrow{\mathfrak{U}} \begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{both } A, B; \\ A \wedge B \end{array} \xrightarrow{\mathfrak{C}} \begin{array}{l} \textbf{dn } \neg\neg A; \\ \textbf{both } A, B \end{array}$$

An alternative form of the same detour is obtained by swapping the order of **dn** and **assume** in the above deduction. We see that *normalize* handles this with the same ease:

$$D = \begin{array}{l}\textbf{assume } A \textbf{ in}\\ \quad \textbf{both } A, B;\\ \textbf{dn } \neg\neg A;\\ \textbf{mp } A \Rightarrow A \wedge B, A\end{array} \xrightarrow{\textit{restructure}} \begin{array}{l}\textbf{assume } A \textbf{ in}\\ \quad \textbf{both } A, B;\\ \textbf{dn } \neg\neg A;\\ \textbf{both } A, B;\\ \textbf{mp } A \Rightarrow A \wedge B, A\end{array} \xrightarrow{\mathfrak{P}}$$

$$\begin{array}{l}\textbf{assume } A \textbf{ in}\\ \quad \textbf{both } A, B;\\ \textbf{dn } \neg\neg A;\\ \textbf{both } A, B;\\ A \wedge B\end{array} \xrightarrow{\mathfrak{U}} \begin{array}{l}\textbf{dn } \neg\neg A;\\ \textbf{both } A, B;\\ A \wedge B\end{array} \xrightarrow{\mathfrak{C}} \begin{array}{l}\textbf{dn } \neg\neg A;\\ \textbf{both } A, B\end{array}$$

We continue with detour (a), based on negation. Letting

$$\mathrm{D} = \begin{array}{l}\textbf{left-and } A \wedge B;\\ \textbf{suppose-absurd } \neg A \textbf{ in}\\ \quad \textbf{absurd } A, \neg A;\\ \textbf{dn } \neg\neg A\end{array}$$

and recalling our desugaring of **suppose-absurd** into a composition of **assume** and Modus Tollens, we have:

$$D = \begin{array}{l}\textbf{left-and } A \wedge B;\\ \textbf{begin}\\ \quad \textbf{assume } \neg A \textbf{ in}\\ \quad\quad \textbf{absurd } A, \neg A;\\ \quad \neg\textbf{false};\\ \quad \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false}\\ \textbf{end};\\ \textbf{dn } \neg\neg A\end{array} \xrightarrow{\textit{restructure}} \begin{array}{l}\textbf{left-and } A \wedge B;\\ \textbf{assume } \neg A \textbf{ in}\\ \quad \textbf{absurd } A, \neg A;\\ \neg\textbf{false};\\ \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false};\\ \textbf{dn } \neg\neg A\end{array} \xrightarrow{\mathfrak{P}}$$

$$\begin{array}{l} \textbf{left-and } A \wedge B; \\ \textbf{assume } \neg A \textbf{ in} \\ \quad \textbf{absurd } A, \neg A; \\ \neg\textbf{false}; \\ \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false}; \\ A \end{array} \xrightarrow{\mathfrak{U}} \begin{array}{l} \textbf{left-and } A \wedge B; \\ A \end{array} \xrightarrow{\mathfrak{C}} \textbf{left-and } A \wedge B$$

A trickier variant of essentially the same detour is shown in the next deduction:

$$\mathrm{D} = \begin{array}{l} \textbf{suppose-absurd } \neg A \textbf{ in} \\ \ \textbf{begin} \\ \quad \textbf{left-and } A \wedge B; \\ \quad \textbf{absurd } A, \neg A \\ \ \textbf{end}; \\ \textbf{dn } \neg\neg A \end{array}$$

Here *normalize* will operate as follows:

$$D = \begin{array}{l} \textbf{begin} \\ \quad \textbf{assume } \neg A \textbf{ in} \\ \qquad \textbf{begin} \\ \qquad\quad \textbf{left-and } A \wedge B; \\ \qquad\quad \textbf{absurd } A, \neg A \\ \qquad \textbf{end}; \\ \quad \neg\textbf{false}; \\ \quad \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false} \\ \textbf{end}; \\ \textbf{dn } \neg\neg A \end{array} \xrightarrow{\textit{restructure}} \begin{array}{l} \textbf{left-and } A \wedge B; \\ \textbf{assume } \neg A \textbf{ in} \\ \quad \textbf{begin} \\ \quad\ \ A; \\ \quad\ \ \textbf{absurd } A, \neg A \\ \quad \textbf{end}; \\ \neg\textbf{false}; \\ \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false}; \\ \textbf{dn } \neg\neg A \end{array} \xrightarrow{\mathfrak{P}}$$

$$\begin{array}{l} \textbf{left-and } A \wedge B; \\ \textbf{assume } \neg A \textbf{ in} \\ \quad \textbf{begin} \\ \quad\ \ A; \\ \quad\ \ \textbf{absurd } A, \neg A \\ \quad \textbf{end}; \\ \neg\textbf{false}; \\ \textbf{mt } \neg A \Rightarrow \textbf{false}, \neg\textbf{false}; \\ A \end{array} \xrightarrow{\mathfrak{U}} \begin{array}{l} \textbf{left-and } A \wedge B; \\ A \end{array} \xrightarrow{\mathfrak{C}} \textbf{left-and } A \wedge B$$

Next we illustrate a disjunction detour. Let D be the following deduction:

dn $\neg\neg(A_1 \wedge B)$;
left-either $A_1 \wedge B, A_2 \wedge B$;
assume $A_1 \wedge B$ **in**
  right-and $A_1 \wedge B$;
assume $A_2 \wedge B$ **in**
  right-and $A_2 \wedge B$;
cd $(A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B$

We have:

$D \xrightarrow{\textit{restructure}}$

dn $\neg\neg(A_1 \wedge B)$;
left-either $A_1 \wedge B, A_2 \wedge B$;
right-and $A_1 \wedge B$;
assume $A_1 \wedge B$ **in**
  B;
assume $A_2 \wedge B$ **in**
  right-and $A_2 \wedge B$;
cd $(A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B$

$\xrightarrow{\mathfrak{P}}$

dn $\neg\neg(A_1 \wedge B)$;
left-either $A_1 \wedge B, A_2 \wedge B$;
right-and $A_1 \wedge B$;
assume $A_1 \wedge B$ **in**
  B;
assume $A_2 \wedge B$ **in**
  right-and $A_2 \wedge B$;
B

$\xrightarrow{\mathfrak{U}}$

dn $\neg\neg(A_1 \wedge B)$;
right-and $A_1 \wedge B$;
B

$\xrightarrow{\mathfrak{C}}$

dn $\neg\neg(A_1 \wedge B)$;
right-and $A_1 \wedge B$

A variant of this detour is contained in

$$\mathrm{D} = \begin{array}{l} \textbf{assume } A_1 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_1 \wedge B; \\ \textbf{assume } A_2 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_2 \wedge B; \\ \textbf{dn } \neg\neg(A_2 \wedge B); \\ \textbf{right-either } A_1 \wedge B, A_2 \wedge B; \\ \textbf{cd } (A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B \end{array}$$

In this case we have:

$$D \xrightarrow{\mathit{restructure}} \begin{array}{l} \textbf{assume } A_1 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_1 \wedge B; \\ \textbf{assume } A_2 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_2 \wedge B; \\ \textbf{dn } \neg\neg(A_2 \wedge B); \\ \textbf{right-and } A_2 \wedge B; \\ \textbf{right-either } A_1 \wedge B, A_2 \wedge B; \\ \textbf{cd } (A_1 \wedge B) \vee (A_2 \wedge B), (A_1 \wedge B) \Rightarrow B, (A_2 \wedge B) \Rightarrow B \end{array} \xrightarrow{\mathfrak{P}}$$

$$\begin{array}{l} \textbf{assume } A_1 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_1 \wedge B; \\ \textbf{assume } A_2 \wedge B \textbf{ in} \\ \quad \textbf{right-and } A_2 \wedge B; \\ \textbf{dn } \neg\neg(A_2 \wedge B); \\ \textbf{right-and } A_2 \wedge B; \\ \textbf{right-either } A_1 \wedge B, A_2 \wedge B; \\ B \end{array} \xrightarrow{\mathfrak{U}} \begin{array}{l} \textbf{dn } \neg\neg(A_2 \wedge B); \\ \textbf{right-and } A_2 \wedge B; \\ B \end{array} \xrightarrow{\mathfrak{C}}$$

$$\begin{array}{l} \textbf{dn } \neg\neg(A_2 \wedge B); \\ \textbf{right-and } A_2 \wedge B \end{array}$$

We close with a biconditional detour. Let D be the following deduction:

$$\begin{array}{l} \textbf{assume } A \wedge B \textbf{ in} \\ \ \textbf{begin} \\ \quad \textbf{left-and } A \wedge B; \\ \quad \textbf{right-and } A \wedge B; \\ \quad \textbf{both } B, A \end{array}$$

  end;
assume $B \wedge A$ **in**
  begin
    right-and $B \wedge A$;
    left-and $B \wedge A$;
    both A, B
  end;
equivalence $A \wedge B \Rightarrow B \wedge A, B \wedge A \Rightarrow A \wedge B$;
left-iff $A \wedge B \Leftrightarrow B \wedge A$

We have:

$D \xrightarrow{\mathfrak{P} \cdot \textit{restructure}}$

assume $A \wedge B$ **in**
  begin
    left-and $A \wedge B$;
    right-and $A \wedge B$;
    both B, A
  end;
assume $B \wedge A$ **in**
  begin
    right-and $B \wedge A$;
    left-and $B \wedge A$;
    both A, B
  end;
equivalence $A \wedge B \Rightarrow B \wedge A, B \wedge A \Rightarrow A \wedge B$;
$A \wedge B \Rightarrow B \wedge A$

$\xrightarrow{\mathfrak{C} \cdot \mathfrak{U}}$

assume $A \wedge B$ **in**
  begin
    left-and $A \wedge B$;
    right-and $A \wedge B$;
    both B, A
  end

6

First-order reasoning in $\mathcal{CND}$

In this chapter we extend $\mathcal{CND}$ to first-order logic with equality. The propositional framework of Chapter 4 is augmented with new syntactic constructs and evaluation semantics that faithfully capture the forms of quantifier reasoning encountered in mathematical practice. We give a variety of examples that demonstrate the readability and writability of the language, and the extent to which formal $\mathcal{CND}$ proofs preserve the structure of informal mathematical arguments. We then develop the theory of first-order $\mathcal{CND}$ deductions in detail, formulating a number of notions and proving several results about the language (conclusion uniqueness, decidability of observational equivalence, etc.). We also show how these results allow the optimization procedures we developed for propositional $\mathcal{CND}$ to carry over directly to the first-order setting without modification. Finally, we work out the Tarskian semantics of predicate $\mathcal{CND}$ and prove that the language is sound and complete for first-order logic with equality.

6.1 Syntax

6.1.1 Terms, formulas, and deductions over logic vocabularies

Let a *logic signature* $\mathbf{\Omega} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ be given, consisting of a set of *constant symbols* $\mathcal{C}$, a set of *function symbols* $\mathcal{F}$, and a set of *relation symbols* $\mathcal{R}$, with each symbol in $\mathcal{F}$ and $\mathcal{R}$ having a unique positive integer associated with it and known as its *arity*. We will assume that these three sets are countable and pairwise disjoint. We also assume that there is a countably infinite set of variables $\mathcal{V}$, disjoint from $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{R}$.[1] The pair $(\mathbf{\Omega}, \mathcal{V})$ will be called a logic *vocabulary*. The equality symbol '=' is reserved and has a special status: we stipulate that it can only occur in a signature as a binary relation symbol. We speak of a *signature with equality* (or without equality) according to whether or not the symbol $=$ is one of the relation symbols. We will use the letters a, b, and c as typical constant symbols; f, g, and h as function symbols; M, P, Q, and R as relation symbols; and x, y, z, u, v, and w as variables. We will use f^n (R^n) to range over function (relation) symbols of arity n.

Note that the constant and function symbols of a logic signature determine a unique *term signature* (in the sense of Appendix A, page 389). Indeed, a logic signature is just a term signature augmented with relation symbols. Thus, whenever it is convenient to do so, we will ignore the relation symbols and treat a given $\mathbf{\Omega}$ as if it were a term signature. Accordingly, the set of *terms* over $\mathbf{\Omega}$ and $\mathcal{V}$, denoted $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$, is defined as usual: all constant symbols c and variables x are terms; and if $t_1, \ldots, t_n$ are terms then so is $f^n(t_1, \ldots, t_n)$—nothing else is a term. The letters s and t will range over terms. See Appendix A for details.

Next, $(\mathbf{\Omega}, \mathcal{V})$-*formulas* (or simply "formulas" when the vocabulary $(\mathbf{\Omega}, \mathcal{V})$ is immaterial) are defined by the following abstract grammar:

$$F ::= R^n(t_1, \ldots, t_n) \mid \mathbf{true} \mid \mathbf{false} \mid \neg F \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid F \Leftrightarrow G \mid (\forall x) F \mid (\exists x) F$$

where R^n ranges over the n-ary relation symbols of $\mathbf{\Omega}$ and $t_1, \ldots, t_n$ are terms over $\mathbf{\Omega}$ and $\mathcal{V}$. Thus formulas have the same structure that propositions did before, only now the atoms are specified in detail: they are expressions of the form $R(t_1, \ldots, t_n)$. In addition, there are two new kinds of formulas, universal and existential quantifications, of the form $(\forall\, x)\, F$ and $(\exists\, x)\, F$, respectively. As before, we will use parentheses (as well as square brackets) to disambiguate parsing; and we will use the letters F, G, and H to denote formulas. For some relation symbols (such as the equality symbol) we will use infix notation. We write $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ for the set of all $(\mathbf{\Omega}, \mathcal{V})$-formulas. By an $(\mathbf{\Omega}, \mathcal{V})$-*assumption base* (or simply an "assumption base" when it is not necessary to mention $(\mathbf{\Omega}, \mathcal{V})$) we will mean a finite subset of $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$. As before, we will use the letter β to range over assumption bases. The letters Φ and Ψ will be used for arbitrary (possibly infinite) subsets of $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$.

Free and bound variable occurences, subformulas, etc., are defined as usual (see Section B.2.1 for details). We write $FV(F)$ ($BV(F)$) for the set of all variables that have free (bound) occurences in F; and we write $Var(F)$ for the set of all variables

[1]Of course we also require disjointness from logical symbols such as $\wedge$, $\forall$, etc., as well as punctuation marks such as the comma and parentheses.

that occur in F. For arbitrary $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ we set

$$FV(\Phi) = \bigcup_{F \in \Phi} FV(F).$$

The class of $\mathcal{CND}$ deductions is now parameterized over a logic vocabulary, and thus we speak of "$(\mathbf{\Omega}, \mathcal{V})$-deductions" (of course in practice we drop the reference to $(\mathbf{\Omega}, \mathcal{V})$ when it is not needed). The abstract syntax of $(\mathbf{\Omega}, \mathcal{V})$-deductions has the same basic structure as before, augmented with four new constructs:

$$\boxed{\begin{array}{lll} D & ::= & F \mid \textit{Prim-Rule}\ F_1, \ldots, F_n \mid \textbf{assume}\ F\ \textbf{in}\ D \mid D_1; D_2 \\ & \mid & \textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t \mid \textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t \\ & \mid & \textbf{pick-any}\ x\ \textbf{in}\ D \mid \textbf{pick-witness}\ x\ \textbf{for}\ (\exists y) F\ \textbf{in}\ D \end{array}}$$

where F, t, and x range over $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$, $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$, and $\mathcal{V}$, respectively, while *Prim-Rule* ranges over the collection of inference rules specified in Figure 4.1. Deductions of the form F are called *claims*; we will also refer to them as *trivial* deductions. A *hypothetical* (or *conditional*) deduction is one of the form **assume** F **in** D; we say that D is the *scope* of the *hypothesis* F. Deductions of the form $D_1; D_2$ are called *composite*, or simply "compositions". The terminology for the remaining cases is:

$$\begin{array}{rll} \textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t: & \textit{universal instantiations} & (\forall\text{-elimination}) \\ \textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t: & \textit{existential generalizations} & (\exists\text{-introduction}) \\ \textbf{pick-any}\ x\ \textbf{in}\ D: & \textit{universal generalizations} & (\forall\text{-introduction}) \\ \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, F\ \textbf{in}\ D: & \textit{existential instantiations} & (\exists\text{-elimination}) \end{array}$$

In the last two forms we refer to D as the *scope* of the *eigenvariable* x. Subdeductions, deduction size, well-formed deductions, etc., are defined as expected (rigorous definitions can be found in Appendix B). Unless we indicate otherwise, the term "deduction" should be understood to mean "well-formed deduction". We write $EV(D)$ for the set of all eigenvariables of D (i.e., the set of all variables x that occur within a subdeduction of D of the form **pick-any** $x \cdots$ or **pick-witness** $x \cdots$), and $Var(D)$ for the set of all variables that occur in D.

Deductions of the form *Prim-Rule* $F_1, \ldots, F_n$, universal specializations,[2] and existential generalizations will be called *primitive* (or *atomic*) *deductions*; all other nontrivial deductions are called *compound*, or *complex*. Deductions "by contradiction", of the form **suppose-absurd** F **in** D, are introduced as syntax sugar in the same manner as before (see Section 4.1). We write $\mathbf{Ded}(\mathbf{\Omega}, \mathcal{V})$ for the set of all $(\mathbf{\Omega}, \mathcal{V})$-deductions.

[2] We will use "specialization" and "instantiation" synonymously.

$$\frac{}{F \approx_\alpha F}\,[A_1] \qquad \frac{F \approx_\alpha G}{\neg F \approx_\alpha \neg G}\,[A_2]$$

$$\frac{F_1 \approx_\alpha G_1 \quad F_2 \approx_\alpha G_2}{(F_1 \circ F_2) \approx_\alpha (G_1 \circ G_2)}\,[A_3] \qquad \frac{\{x \mapsto z\}\, F \approx_\alpha \{y \mapsto z\}\, G}{(Q\,x)\, F \approx_\alpha (Q\,y)\, G}\,[A_4]$$

for $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$. for $Q \in \{\forall, \exists\}$, provided z occurs neither in F nor in G.

Figure 6.1: Definition of the alphabetic equivalence relation $\approx_\alpha$.

6.1.2 Substitutions for formulas and deductions

For any $(\mathbf{\Omega}, \mathcal{V})$-substitution θ, we define a function $\theta^\sharp : \mathbf{Form}(\mathbf{\Omega}, \mathcal{V}) \to \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ as follows:

$$\begin{aligned}
\theta^\sharp(R(t_1, \ldots, t_n)) &= R(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n)) \\
\theta^\sharp(\neg G) &= \neg \theta^\sharp(G) \\
\theta^\sharp(F \circ G) &= \theta^\sharp(F) \circ \theta^\sharp(G) \\
\theta^\sharp((Q\,x)\, G) &= (Q\,x)\, \theta[x \mapsto x]^\sharp(G)
\end{aligned}$$

for $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, $Q \in \{\forall, \exists\}$. We refer to $\theta^\sharp(F)$ as *the result of applying θ to F*. To simplify notation, we will usually write $\theta\, F$ as an abbreviation for $\theta^\sharp(F)$. For a set of formulas Φ we write $\theta\, \Phi$ as a shorthand for $\{\theta^\sharp(F) \mid F \in \Phi\}$.

A substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ can be *safely applied* to a formula F if no x_i occurs free in a subformula of F of the form $(Q\,x)\, G$ (for some quantifier Q), where $x \in \mathit{Var}(t_i)$. This can always be ensured by consistently renaming the bound variables of F so as to obtain an *alphabetic variant* of F whose bound variables are disjoint from $\mathit{Var}(t_i)$, $i = 1, \ldots, n$. Two formulas are alphabetic variants—or *alphabetically equivalent*—when they differ only in the names of their bound variables. The relation of alphabetic equivalence is denoted by $\approx_\alpha$; it is precisely defined in Figure 6.1. The reader will verify that $\approx_\alpha$ is an equivalence relation. From now on alphabetically equivalent formulas will be identified. That is, two formulas will be considered identical iff they are alphabetic variants. Thus we will essentially be working with equivalence classes of formulas. The following simple result will come handy later:

Lemma 6.1 *If $y \notin \mathit{Var}(F)$ then $(Q\,x)\, F \approx_\alpha (Q\,y)\, \{x \mapsto y\}\, F$ (for $Q \in \{\forall, \exists\}$).*

$$
\begin{array}{rcl}
\theta^\star(F) & = & \theta\, F \\
\theta^\star(\textit{Prim-Rule } F_1, \ldots, F_n) & = & \textit{Prim-Rule } \theta\, F_1, \ldots, \theta\, F_n \\
\theta^\star(\textbf{specialize } F \textbf{ with } t) & = & \textbf{specialize } \theta\, F \textbf{ with } \overline{\theta}(t) \\
\theta^\star(\textbf{ex-generalize } F \textbf{ from } t) & = & \textbf{ex-generalize } \theta\, F \textbf{ from } \overline{\theta}(t) \\
\theta^\star(\textbf{assume } F \textbf{ in } D) & = & \textbf{assume } \theta\, F \textbf{ in } \theta^\star(D) \\
\theta^\star(D_1; D_2) & = & \theta^\star(D_1); \theta^\star(D_2) \\
\theta^\star(\textbf{pick-any } x \textbf{ in } D) & = & \textbf{pick-any } x \textbf{ in } \theta[x \mapsto x]^\star(D) \\
\theta^\star(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D) & = & \textbf{pick-witness } x \textbf{ for } \theta\, F \textbf{ in } \theta[x \mapsto x]^\star(D)
\end{array}
$$

Figure 6.2: Definition of $\theta^\star(D)$.

Proof: Consider any variable z that does not occur in F. Since y does not occur in F, Lemma 6.17 below entails the identity $\{y \mapsto z\}\,\{x \mapsto y\}\,F = \{x \mapsto z\}\,F$. Therefore, by $[A_1]$,

$$\{x \mapsto z\}\,F \approx_\alpha \{y \mapsto z\}\,\{x \mapsto y\}\,F$$

and by $[A_4]$, $(Q\,x)\,F \approx_\alpha (Q\,y)\,\{x \mapsto y\}\,F$. ■

Finally, for any $(\mathbf{\Omega}, \mathcal{V})$-substitution θ we define a function

$$\theta^\star : \mathbf{Ded}(\mathbf{\Omega}, \mathcal{V}) \to \mathbf{Ded}(\mathbf{\Omega}, \mathcal{V})$$

as shown in Figure 6.2. We refer to $\theta^\star(D)$ as the result of *applying θ to D*. As we did with applications of substitutions to formulas, we will write $\theta\, D$ as an abbreviation for $\theta^\star(D)$. Intuitively, $\theta\, D$ is obtained by replacing every free occurence of all variables x in D by $\theta(x)$. Note that in deductions of the form

$$\textbf{pick-any } x \textbf{ in } D' \tag{6.1}$$

and

$$\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D' \tag{6.2}$$

the eigenvariable x is considered bound, and thus there are no free occurences of it to be replaced. Accordingly, when we come to apply θ to the body D', we “mask” whatever value θ prescribes for x by extending θ with the binding $[x \mapsto x]$, which ensures that all free occurences of x inside D' will remain unchanged. (Observe, however, that in existential instantiations of the form 6.2 the scope of x does *not* include F; that is why in the definition in Figure 6.2 we apply θ—rather than $\theta[x \mapsto x]$—to F.)

The fact that eigenvariables are bound and have scope also raises the specter of variable capture. Consider again a deduction of the form 6.1 or 6.2 and suppose that a variable $y \neq x$ occurs free in D', where x has occurences in $\theta(y)$. Then if we apply θ without precaution and replace y inside D' by $\theta(y)$, the said occurences of x will become captured. A condition that rules out this possibility is the following: no eigenvariable should occur in $\theta(y)$, for all $y \in \mathit{Supp}(\theta)$. Accordingly, we will say that a substitution θ is *safe* for a deduction D iff

$$\mathit{RanVar}(\theta) \cap \mathit{EV}(D) = \emptyset. \tag{6.3}$$

In fact this condition is coarser than necessary, but has the advantage of simplifying certain proofs. Later on we will see that deductions that differ only in the names of their eigenvariables are essentially identical (observationally equivalent), and hence we can always rename the eigenvariables of a deduction to our liking. For this reason we may tacitly assume that any substitution θ is safe for a given D: we simply rename the latter's eigenvariables away from $\mathit{RanVar}(\theta)$. But until we get to that point we will need to use the notion of safety given by 6.3.

6.2 Evaluation semantics

Before delving into the formal semantics we will discuss the evaluation of the new constructs informally. Let us first consider the two new primitive rules, universal specializations, of the form

$$\textbf{specialize } (\forall\, x)\, F \textbf{ with } t \tag{6.4}$$

and existential generalizations, of the form

$$\textbf{ex-generalize } (\exists\, x)\, F \textbf{ from } t. \tag{6.5}$$

For 6.4, if the formula $(\forall\, x)\, F$ is in the assumption base, then the result is the formula $\{x \mapsto t\}\, F$; it is an error if $(\forall\, x)\, F$ is not in the assumption base. It should be intuitively clear that this is a sound rule. For 6.5, if the formula $\{x \mapsto t\}\, F$ is in the assumption base (so that t is a "witness" for the existential claim $(\exists x)F$), then the generalization $(\exists x)F$ is returned; otherwise an error occurs. It should be clear that this rule, too, is sound (we will formally prove the soundness of both rules in Section 6.6).

Next, let us see how **pick-any** ($\forall$–introduction) and **pick-witness** ($\exists$–elimination) work. To evaluate a deduction **pick-any** x **in** D in some assumption base β, we simply evaluate D in β, in a context in which x refers to an *arbitrary* individual. If D produces a formula F, then the result of the whole deduction is the universal generalization $(\forall x)F$. It should be clear that if x does not occur free in any member of β then it already refers to an "arbitrary individual" and we do not have to do anything special: we just evaluate D in β, and if that results in some F, we return $(\forall x)F$. In general, however, x might occur free in β, and in that case this method would be unsound. For instance, evaluating

$$\textbf{pick-any } x \textbf{ in } x = 0$$

in an assumption base that contained the formula $x = 0$ would result in $(\forall x)x = 0$.

There are several ways to formally enforce the requirement that x be "arbitrary", although we will eventually see that the most elegant and efficient solution requires a more sophisticated syntax and semantics such as found in the $\lambda\mu$-calculus (see Section 9.7). But there are alternatives in the present setting as well, two of which are:

1. To evaluate **pick-any** x **in** D in β, evaluate D in the subset of β comprised by those formulas which do not contain free occurences of x. Thus we explicitly "block out" any special knowledge we might have about x during the evaluation of D. If F is thus obtained, return $(\forall x)F$.

2. To evaluate **pick-any** x **in** D in β, evaluate D' in β, where D' is obtained from D by replacing every free occurence of x by some fresh variable y (one that does not occur in β or in D). If F is thus obtained, return $(\forall y)F$.

Both approaches are sound but the first one destroys the monotonicity aesthetic, and for that reason we will opt for the second alternative. From a practical standpoint this should be irrelevant to the user: as far as writing deductions is concerned, all that matters is that in deductions of the form **pick-any** x **in** D the system will ensure that no special assumptions about x are made.

Finally we consider the evaluation of deductions of the form

$$\textbf{pick-witness } x \textbf{ for } (\exists y)F \textbf{ in } D$$

in some β. The idea here is that the existential generalization $(\exists y)F$ is already established (i.e., in β), and we are simply picking x as a witness term for it in the course of D. We thus evaluate D in $\beta \cup \{\{y \mapsto x\}\, F\}$; if that results in a conclusion G, we return G. That is, the result of the entire deduction is the result of evaluating D in $\beta \cup \{\{y \mapsto x\}\, F\}$. Loosely speaking, the reasoning is: "We know that there exists an individual for which F holds. Let the name x refer to that individual. If with this convention we can prove (via D) some formula G, then we are entitled to infer G".

Of course there are some caveats to be observed. First, no special assumptions about x should be made. The variable x should only serve as a "dummy"—the deduction should go through if we replace x by any other variable. It is not difficult to see what could go astray otherwise. Suppose we know that x is even, i.e., the formula $Even(x)$ is in β. And suppose that we know that there exists an odd number, $(\exists y)\, Odd(y)$. If we let x be the witness for this claim, we could deduce that there is a number that is both odd and even. In our case, the phrase "no special assumptions about x" simply means that x should not occur free in β. As in the case of $\forall$–introduction, the semantics will ensure that this requirement is satisfied by replacing x throughout by some fresh variable z that does not occur in β or in D. Thus the burden of having to keep track of which names are currently in use and can or cannot be used as witnesses shifts from the user to the system. The user can choose any name he pleases as a witness, even if it occurs free in the assumption base. Since the name is meant to be used as a "dummy", the system will automatically replace it by a fresh name throughout, call it z.

For the same reasons, we will also require that the fresh variable z should not occur free in the conclusion G. The idea is that, because x is a dummy, the replacement z should be arbitrary as long as it is fresh; it should not affect the conclusion G. But this does not hold if one fresh variable z results in a conclusion such as $R(z)$ and another fresh variable z' results in a conclusion $R(z')$. The conclusions $R(z)$ and $R(z')$ are *not* the same (i.e., not α-convertible), due to the free occurences of z and z'.

We stress that the eigenvariable restrictions are *relegated to the semantics*, and hence, from a practical standpoint, to the implementation of the language. The *user* is free from all the tedious book-keeping that is usually necessary for quantifier manipulation in formal proof systems. This is part of the reason why, as the examples will show shortly, proofs in this language mirror informal mathematical reasoning so closely.

Figure 6.3 depicts the formal evaluation semantics of first-order $\mathcal{CND}$. Rules $[R_1]$—$[R_5]$ are just as in the propositional case. Rules $[R_6]$—$[R_9]$ cover the new syntax forms introduced in this chapter, formalizing the ideas we discussed above. In addition to these rules, we also need to postulate the axioms shown in Figure 4.3, which prescribe the behavior of the primitive inference rules for propositional reasoning. Those axioms carry over without change—save the obvious difference that we are now dealing with formulas rather than propositions—and so we will not repeat them here.

Now fix a logic vocabulary $(\boldsymbol{\Omega}, \mathcal{V})$. As before, we write $\beta \vdash D \leadsto F$ to mean that there is a finite sequence of judgments

$$\beta_1 \vdash D_1 \leadsto F_1, \ldots, \beta_n \vdash D_n \leadsto F_n \tag{6.6}$$

such that $\beta_n = \beta, D_n = D, F_n = F$, and where every judgment $\beta_i \vdash D_i \leadsto F_i$ in 6.6 is

$$\frac{}{\beta \vdash \textbf{true} \leadsto \textbf{true}}\ [R_1] \qquad \frac{}{\beta \vdash \neg\textbf{false} \leadsto \neg\textbf{false}}\ [R_2]$$

$$\frac{}{\beta \cup \{F\} \vdash F \leadsto F}\ [R_3] \qquad \frac{\beta \cup \{F\} \vdash D \leadsto G}{\beta \vdash \textbf{assume}\ F\ \textbf{in}\ D \leadsto F \Rightarrow G}\ [R_4]$$

$$\frac{\beta \vdash D_1 \leadsto F_1 \qquad \beta \cup \{F_1\} \vdash D_2 \leadsto F_2}{\beta \vdash D_1; D_2 \leadsto F_2}\ [R_5]$$

$$\frac{}{\beta \cup \{(\forall\, x)\, F\} \vdash \textbf{specialize}\ (\forall\, x)\, F \textbf{with}\ t \leadsto \{x \mapsto t\}\, F}\ [R_6]$$

$$\frac{}{\beta \cup \{\{x \mapsto t\}\, F\} \vdash \textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t \leadsto (\exists\, x)\, F}\ [R_7]$$

$$\frac{\beta \vdash \{x \mapsto y\}\, D \leadsto F}{\beta \vdash \textbf{pick-any}\ x\ \textbf{in}\ D \leadsto (\forall y) F}\ [R_8]$$

whenever y does not occur in β or in D.

$$\frac{\beta \cup \{(\exists\, y)\, F, \{y \mapsto z\}\, F\} \vdash \{x \mapsto z\}\, D \leadsto G}{\beta \cup \{(\exists\, y)\, F\} \vdash \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, F\ \textbf{in}\ D \leadsto G}\ [R_9]$$

whenever z does not occur in $\beta \cup \{(\exists\, y)\, F\}$ or in D, and $z \notin FV(G)$.

Figure 6.3: Evaluation semantics of predicate $\mathcal{CND}$.

either an instance of $[R_1]$, $[R_2]$, $[R_3]$, or one of the primitive-rule axioms of Figure 4.3; or else follows from previous judgments through one of the rules $[R_4]$—$[R_9]$. For an arbitrary set of formulas Φ, we write $\Phi \vdash_{\mathcal{CND}} F$ to signify that there is a deduction D and an assumption base $\beta \subseteq \Phi$ such that $\beta \vdash D \leadsto F$. The following two results are immediate consequences of the foregoing definitions, just as in the propositional version:

Lemma 6.2 (Reflexivity) *$\Phi \vdash_{\mathcal{CND}} F$ for all $F \in \Phi$.*

Lemma 6.3 (Monotonicity) *If $\Phi \vdash_{\mathcal{CND}} F$ then $\Phi \cup \Psi \vdash_{\mathcal{CND}} F$.*

$$\frac{}{\beta \vdash \textbf{ref}\ t \leadsto t = t}\ [\mathit{Ref}]$$

$$\frac{}{\beta \cup \{s = t\} \vdash \textbf{leibniz}\ \ F\ \ \textbf{with}\ \ x, s, t \leadsto \{x \mapsto s\}\, F \Leftrightarrow \{x \mapsto t\}\, F}\ [\mathit{Leibniz}]$$

Figure 6.4: Evaluation semantics for the primitive equality rules.

Equality

Special primitive inference rules are needed to handle signatures with equality. The number of such rules can be reduced to a minimum of two: reflexivity and Leibniz's rule. All of the other familiar properties of equality (symmetry, transitivity, congruence, etc.) can be derived from these two rules. Our formulation of these rules is as follows:

- *Reflexivity*: Applications of this rule are of the form

  $$\textbf{ref}\ \ t$$

  for an arbitrary term t. This simply produces the result $t = t$, regardless of the contents of the assumption base.

- *Leibniz*: Applications of this rule are of the form

  $$\textbf{leibniz}\ \ F\ \ \textbf{with}\ \ x, s, t$$

  for any formula F, variable x and terms s and t. If the equality $s = t$ is in the assumption base then the result is the biconditional

  $$\{x \mapsto s\}\, F \Leftrightarrow \{x \mapsto t\}\, F.$$

  If $s = t$ does not hold (i.e., is not in the assumption base), then an error occurs.

The formal semantics of these two rules are shown in Figure 6.4.

Derived inference rules for dealing with equality can easily be obtained from [*Ref*] and [*Leibniz*] in such a way that every application of a derived inference rule can be mechanically replaced by a deduction that uses only the primitive inference rules. (Later we will develop a more uniform way of formulating "derived" rules via *methods*.) We discuss a few such derived rules and rule schemas below:

1. *Symmetry*: From $s = t$ we may infer $t = s$. We model this with the rule **swap**, which is applied as follows:

   swap $s = t$

   for any terms s and t. The idea is that if $s = t$ is in the assumption base, then the result $t = s$ should be produced; otherwise an error should occur. This can be achieved by "desugaring" every application of **swap** of the form shown above into the following deduction:

   leibniz $t = x$ **with** x, s, t;
   ;; This gives $t = s \Leftrightarrow t = t$.
   right-iff $t = s \Leftrightarrow t = t$;
   ;; Now we have $t = t \Rightarrow t = s$.
   ref t;
   ;; And $t = t$ from reflexivity.
   modus-ponens $t = t \Rightarrow t = s, t = t$
   ;; Finally we get $t = s$.

   for any variable $x \notin Var(s) \cup Var(t)$.

2. *Transitivity*: From $s_1 = s_2$ and $s_2 = s_3$ we may infer $s_1 = s_3$. We model this with a rule called **tran**, which is applied as follows:

   tran $s_1 = s_2, s_2 = s_3$

   for any terms s_1, s_2 and s_3. The intended semantics are: if $s_1 = s_2$ and $s_2 = s_3$ are in the assumption base, then $s_1 = s_3$ should be produced; otherwise an error should occur. This can be achieved with the following desugaring:

   leibniz $x = s_3$ **with** x, s_1, s_2;
   ;; This gives $s_1 = s_3 \Leftrightarrow s_2 = s_3$.
   right-iff $s_1 = s_3 \Leftrightarrow s_2 = s_3$;
   ;; Now we have $s_2 = s_3 \Rightarrow s_1 = s_3$.
   modus-ponens $s_2 = s_3 \Rightarrow s_1 = s_3, s_2 = s_3$
   ;; Finally we get $s_1 = s_3$.

   for any x that does not occur in s_1, s_2, or s_3.

3. *Congruence*: If we have $s_1 = t_1, \ldots, s_n = t_n$ and $R(s_1, \ldots, s_n)$, for any n-ary relation symbol R, then we may infer $R(t_1, \ldots, t_n)$. This is a rule *schema*, with one instance for each different value of n. The general schema is

$$\textbf{cong}\ \ R(t_1, \ldots, t_n)\ \ \textbf{from}\ R(s_1, \ldots, s_n), s_1 = t_1, \ldots, s_n = t_n$$

the intended semantics being that if the equalities $s_1 = t_1, \ldots, s_n = t_n$ and the formula $R(s_1, \ldots, s_n)$ are in the assumption base, then the formula $R(t_1, \ldots, t_n)$ is derived; otherwise an error occurs. We show how to desugar applications of this rule for $n = 2$; the idea is the same for all other values of n.

leibniz $R(x, s_2)$ **with** x, s_1, t_1;
;; This gives $R(s_1, s_2) \Leftrightarrow R(t_1, s_2)$.
left-iff $R(s_1, s_2) \Leftrightarrow R(t_1, s_2)$;
;; Now we have $R(s_1, s_2) \Rightarrow R(t_1, s_2)$.
modus-ponens $R(s_1, s_2) \Rightarrow R(t_1, s_2), R(s_1, s_2)$;
;; We now have $R(t_1, s_2)$.
leibniz $R(t_1, x)$ **with** x, s_2, t_2;
;; This gives $R(t_1, s_2) \Leftrightarrow R(t_1, t_2)$.
left-iff $R(t_1, s_2) \Leftrightarrow R(t_1, t_2)$;
;; Now we have $R(t_1, s_2) \Rightarrow R(t_1, t_2)$.
modus-ponens $R(t_1, s_2) \Rightarrow R(t_1, t_2), R(t_1, s_2)$
;; And finally we get $R(t_1, t_2)$.

(for $x \notin Var(s_i) \cup Var(t_i), i = 1, \ldots, n$.)

The same method can be used for congruences involving function symbols, i.e., the derivation of formulas $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ from the equalities $s_1 = t_1$, $\ldots$, $s_n = t_n$. The same rule schema, **cong**, can be used for such derivations, as in

$$\textbf{cong}\ \ f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)\ \ \textbf{from}\ s_1 = t_1, \ldots, s_n = t_n$$

6.3 Examples

Our first example derives $\neg(\exists\, x)\, \neg P(x)$ from the assumption $(\forall\, x)\, P(x)$ (this is one of a family of inference rules collectively known as *quantifier negation*):

assert $(\forall\, x)\, P(x)$;
suppose-absurd $(\exists\, x)\, \neg P(x)$ **in**

pick-witness y **for** $(\exists\, x)\, \neg P(x)$ **in**
  begin
    specialize $(\forall\, x)\, P(x)$ **with** y;
    absurd $P(y), \neg P(y)$
  end

Our second example infers $(\forall\, y)\, R(y)$ from $(\forall\, x)\, [P(x) \wedge Q(x)]$ and $(\forall\, x)\, [Q(x) \Rightarrow R(x)]$:

assert $(\forall\, x)\, [P(x) \wedge Q(x)]$;
assert $(\forall\, x)\, [Q(x) \Rightarrow R(x)]$;
pick-any y **in**
  begin
    specialize $(\forall\, x)\, [P(x) \wedge Q(x)]$ **with** y;
    right-and $P(y) \wedge Q(y)$;
    specialize $(\forall\, x)\, [Q(x) \Rightarrow R(x)]$ **with** y;
    modus-ponens $Q(y) \Rightarrow R(y), Q(y)$
  end

Our third example derives the tautology $(\forall\, x)\, [P(y) \Rightarrow Q(x)] \Rightarrow [P(y) \Rightarrow (\forall\, z)\, Q(z)]$:

assume $(\forall\, x)\, [P(y) \Rightarrow Q(x)]$ **in**
  assume $P(y)$ **in**
    pick-any z **in**
      begin
        specialize $(\forall\, x)\, [P(y) \Rightarrow Q(x)]$ **with** z;
        modus-ponens $P(y) \Rightarrow Q(z), P(y)$
      end

Observe how our choice of which language construct to use at each step is dictated by the main logical connective of the formula we are trying to prove—a hallmark of natural deduction. At the top level we are trying to establish a conditional $F \Rightarrow G$, so we start out with a skeleton deduction **assume** F **in** D, where D is to establish G (perhaps with the help of the hypothesis F). Now G itself is a conditional $G_1 \Rightarrow G_2$, so D becomes refined into another hypothetical deduction of the form **assume** G_1 **in** D', where D', yet to be discovered, should establish G_2, i.e., $(\forall z)Q(z)$. Universal generalizations are introduced by **pick-any** deductions, so this is the form that D' assumes. The details are then filled in with elementary forward reasoning.

Our fourth example derives $(\exists\, y)\, [Q(y) \wedge R(y)]$ from the set of premises

$$\{(\forall\, x)\, [P(x) \Rightarrow Q(x)], (\exists\, x)\, [R(x) \wedge P(x)]\} :$$

assert $(\forall\, x)\,[P(x) \Rightarrow Q(x)]$;
assert $(\exists\, x)\,[R(x) \wedge P(x)]$;
pick-witness z **for** $(\exists\, x)\,[R(x) \wedge P(x)]$ **in**
  begin
    right-and $R(z) \wedge P(z)$;
    specialize $(\forall\, x)\,[P(x) \Rightarrow Q(x)]$ **with** z;
    modus-ponens $P(z) \Rightarrow Q(z), P(z)$;
    left-and $R(z) \wedge P(z)$;
    both $Q(z), R(z)$;
    ex-generalize $(\exists\, y)\,[Q(y) \wedge R(y)]$ **from** z
  end

Our next example is taken from an exercise in Copi's "Symbolic Logic" [17]. The exercise asks for the formalization of the following argument:

> All the accused are guilty. All who are convicted will hang. Therefore, if all who are guilty are convicted, then all the accused will hang.

Using $A(x), C(x), G(x)$, and $H(x)$, respectively, for "x is accused", "x is convicted", "x is guilty", and "x will hang", we are asked to derive

$$(\forall\, x)\,[G(x) \Rightarrow C(x)] \Rightarrow (\forall\, y)\,[A(y) \Rightarrow H(y)]$$

from $(\forall\, x)\,[A(x) \Rightarrow G(x)]$ and $(\forall\, x)\,[C(x) \Rightarrow H(x)]$. The following deduction accomplishes this:

assert $(\forall\, x)\,[A(x) \Rightarrow G(x)]$;
assert $(\forall\, x)\,[C(x) \Rightarrow H(x)]$;
assume $(\forall\, x)\,[G(x) \Rightarrow C(x)]$ **in**
  pick-any y **in**
    assume $A(y)$ **in**
      begin
        specialize $(\forall\, x)\,[A(x) \Rightarrow G(x)]$ **with** y;
        modus-ponens $A(y) \Rightarrow G(y), A(y)$;
        specialize $(\forall\, x)\,[G(x) \Rightarrow C(x)]$ **with** y;
        modus-ponens $G(y) \Rightarrow C(y), G(y)$;
        specialize $(\forall\, x)\,[C(x) \Rightarrow H(x)]$ **with** y;
        modus-ponens $C(y) \Rightarrow H(y), C(y)$
      end

Finally, we present additional $\mathcal{CND}$ proofs of some well-known and important tautologies of predicate logic. We hope these will serve as evidence that this language captures common mathematical reasoning in a formal but *fluid* style. The reader should glance at a few of these proofs and then try the rest on his own. We believe that the reader will be pleasantly surprised by how easy it is to write (and read!) deductions in such a language.

$$\boxed{(\forall\, x)\, [P(x) \wedge Q(x)] \Rightarrow (\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x)}$$

Proof:

assume $(\forall\, x)\, [P(x) \wedge Q(x)]$ **in**
  begin
    pick-any y **in**
      begin
        specialize $(\forall\, x)\, [P(x) \wedge Q(x)]$ **with** y;
        left-and $P(y) \wedge Q(y)$
      end;
    pick-any y **in**
      begin
        specialize $(\forall\, x)\, [P(x) \wedge Q(x)]$ **with** y;
        right-and $P(y) \wedge Q(y)$
      end;
    both $(\forall\, y)\, P(y), (\forall\, y)\, Q(y)$
  end

$$\boxed{[(\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x)] \Rightarrow (\forall\, x)\, [P(x) \wedge Q(x)]}$$

Proof:

assume $(\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x)$ **in**
  pick-any y **in**
    begin
      left-and $(\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x)$;
      specialize $(\forall\, x)\, P(x)$ **with** y;
      right-and $(\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x)$;
      specialize $(\forall\, x)\, Q(x)$ **with** y;
      both $P(y), Q(y)$
    end

$$\boxed{(\exists\, x)\, [P(x) \wedge Q(x)] \Rightarrow (\exists\, x)\, P(x) \wedge (\exists\, x)\, Q(x)}$$

Proof:
  assume $(\exists\, x)\, [P(x) \wedge Q(x)]$ **in**
    pick-witness y **for** $(\exists\, x)\, [P(x) \wedge Q(x)]$ **in**
      begin
        left-and $P(y) \wedge Q(y)$;
        ex-generalize $(\exists\, x)\, P(x)$ **from** y;
        right-and $P(y) \wedge Q(y)$;
        ex-generalize $(\exists\, x)\, Q(x)$ **from** y;
        both $(\exists\, x)\, P(x), (\exists\, x)\, Q(x)$
      end

$$\boxed{[(\forall\, x)\, P(x) \vee (\forall\, x)\, Q(x)] \Rightarrow (\forall\, x)\, [P(x) \vee Q(x)]}$$

Proof:
  assume $(\forall\, x)\, P(x) \vee (\forall\, x)\, Q(x)$ **in**
    pick-any y **in**
      begin
        assume $(\forall\, x)\, P(x)$ **in**
          begin
            specialize $(\forall\, x)\, P(x)$ **with** y;
            left-either $P(y), Q(y)$
          end;
        assume $(\forall\, x)\, Q(x)$ **in**
          begin
            specialize $(\forall\, x)\, Q(x)$ **with** y;
            right-either $P(y), Q(y)$
          end;
        cd $(\forall\, x)\, P(x) \vee (\forall\, x)\, Q(x)$, $(\forall\, x)\, P(x) \Rightarrow P(y) \vee Q(y)$,
            $(\forall\, x)\, Q(x) \Rightarrow P(y) \vee Q(y)$
      end

$$\boxed{\neg(\exists\, x)\, P(x) \Rightarrow (\forall\, x)\, \neg P(x)}$$

Proof:
  assume $\neg(\exists\, x)\, P(x)$ **in**
    pick-any y **in**

suppose-absurd $P(y)$ **in**
 begin
  ex-generalize $(\exists x)\,P(x)$ **from** y;
  absurd $(\exists x)\,P(x), \neg(\exists x)\,P(x)$
 end

$$\boxed{(\forall x)\,\neg P(x) \Rightarrow \neg(\exists x)\,P(x)}$$

Proof:
assume $(\forall x)\,\neg P(x)$ **in**
 suppose-absurd $(\exists x)\,P(x)$ **in**
  pick-witness y **for** $(\exists x)\,P(x)$ **in**
   begin
    specialize $(\forall x)\,\neg P(x)$ **with** y;
    absurd $P(y), \neg P(y)$
   end

$$\boxed{\neg(\forall x)\,P(x) \Rightarrow (\exists x)\,\neg P(x)}$$

Proof:
assume $\neg(\forall x)\,P(x)$ **in**
 begin
  suppose-absurd $\neg(\exists x)\,\neg P(x)$ **in**
   begin
    pick-any y **in**
     begin
      suppose-absurd $\neg P(y)$ **in**
       begin
        ex-generalize $(\exists x)\,\neg P(x)$ **from** y;
        absurd $(\exists x)\,\neg P(x), \neg(\exists x)\,\neg P(x)$
       end;
      dn $\neg\neg P(y)$;
     end;
    absurd $(\forall x)\,P(x), \neg(\forall x)\,P(x)$
   end;
  dn $\neg\neg(\exists x)\,\neg P(x)$
 end

$$\boxed{(\exists\, x)\, \neg P(x) \Rightarrow \neg(\forall\, x)\, P(x)}$$

Proof:

assume $(\exists\, x)\, \neg P(x)$ **in**
  suppose-absurd $(\forall\, x)\, P(x)$ **in**
    pick-witness y **for** $(\exists\, x)\, \neg P(x)$ **in**
      begin
        specialize $(\forall\, x)\, P(x)$ **with** y;
        absurd $P(y), \neg P(y)$
      end

$$\boxed{(\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x) \Rightarrow (\exists\, x)\, [P(x) \vee Q(x)]}$$

Proof:

assume $(\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)$ **in**
  begin
    assume $(\exists\, x)\, P(x)$ **in**
      pick-witness y **for** $(\exists\, x)\, P(x)$ **in**
      begin
        left-either $P(y) \vee Q(y)$;
        ex-generalize $(\exists\, x)\, [P(x) \vee Q(x)]$ **from** y
      end;
    assume $(\exists\, x)\, Q(x)$ **in**
      pick-witness y **for** $(\exists\, x)\, Q(x)$ **in**
      begin
        right-either $P(y) \vee Q(y)$;
        ex-generalize $(\exists\, x)\, [P(x) \vee Q(x)]$ **from** y
      end;
    cd $(\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)$, $(\exists\, x)\, P(x) \Rightarrow (\exists\, x)\, [P(x) \vee Q(x)]$,
        $(\exists\, x)\, Q(x) \Rightarrow (\exists\, x)\, [P(x) \vee Q(x)]$
  end

$$\boxed{(\exists\, x)\, [P(x) \vee Q(x)] \Rightarrow (\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)}$$

Proof:

assume $(\exists\, x)\, [P(x) \vee Q(x)]$ **in**
  pick-witness y **for** $(\exists\, x)\, [P(x) \vee Q(x)]$ **in**
    begin

assume $P(y)$ **in**
  begin
    ex-generalize $(\exists\, x)\, P(x)$ **from** y;
    left-either $(\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)$
  end;
assume $Q(y)$ **in**
  begin
    ex-generalize $(\exists\, x)\, Q(x)$ **from** y;
    right-either $(\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)$
  end;
cd $P(y) \vee Q(y), P(y) \Rightarrow (\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x),$
    $Q(y) \Rightarrow (\exists\, x)\, P(x) \vee (\exists\, x)\, Q(x)$
end

$$\boxed{(\forall\, x)\, [P(x) \Rightarrow Q(x)] \Rightarrow (\forall\, x)\, P(x) \Rightarrow (\forall\, x)\, Q(x)}$$

Proof:
assume $(\forall\, x)\, [P(x) \Rightarrow Q(x)]$ **in**
  assume $(\forall\, x)\, P(x)$ **in**
    pick-any y **in**
      begin
        specialize $(\forall\, x)\, [P(x) \Rightarrow Q(x)]$ **with** y;
        specialize $(\forall\, x)\, P(x)$ **with** y;
        modus-ponens $P(y) \Rightarrow Q(y), P(y)$
      end

$$\boxed{(\forall\, x)\, [P(x) \Rightarrow Q(x)] \Rightarrow (\exists\, x)\, P(x) \Rightarrow (\exists\, x)\, Q(x)}$$

Proof:
assume $(\forall\, x)\, [P(x) \Rightarrow Q(x)]$ **in**
  assume $(\exists\, x)\, P(x)$ **in**
    pick-witness y **for** $(\exists\, x)\, P(x)$ **in**
      begin
        specialize $(\forall\, x)\, [P(x) \Rightarrow Q(x)]$ **with** y;
        modus-ponens $P(y) \Rightarrow Q(y), P(y)$;
        ex-generalize $(\exists\, x)\, Q(x)$ **from** y
      end

$$\boxed{(\forall\, x)\,[P(x) \Rightarrow Q(x)] \Rightarrow (\forall\, x)\, P(x) \Rightarrow (\exists\, x)\, Q(x)}$$

Proof:

assume $(\forall\, x)\,[P(x) \Rightarrow Q(x)]$ **in**
  assume $(\forall\, x)\, P(x)$ **in**
    begin
      suppose-absurd $\neg(\exists\, x)\, Q(x)$ **in**
        begin
          $(\forall\, x)\,\neg Q(x)$;   (Insert the deduction of $(\forall\, x)\,\neg Q(x)$ from $\neg(\exists\, x)\, Q(x)$).
          pick-any y **in**
            begin
              specialize $(\forall\, x)\,[P(x) \Rightarrow Q(x)]$ **with** y;
              specialize $(\forall\, x)\, P(x)$ **with** y;
              modus-ponens $P(y) \Rightarrow Q(y), P(y)$;
              specialize $(\forall\, x)\,\neg Q(x)$ **with** y;
              absurd $Q(y), \neg Q(y)$
            end;
          specialize $(\forall\, y)$ **false with** y;
        end;
      dn $\neg\neg(\exists\, x)\, Q(x)$
    end

$$\boxed{(\forall\, x)\,(\exists\, y)\, P(y) \Rightarrow P(x)}$$

Proof:

pick-any x **in**
  begin
    assume $P(x)$ **in**
      $P(x)$;
    ex-generalize $(\exists\, y)\, P(y) \Rightarrow P(x)$ **from** x
  end

$$\boxed{(\forall\, x)\,(\forall\, y)\, P(x,y) \Rightarrow (\forall\, y)\,(\forall\, x)\, P(x,y)}$$

Proof:

assume $(\forall\, x)\,(\forall\, y)\, P(x,y)$ **in**
  pick-any z **in**
    pick-any w **in**

begin
  specialize $(\forall x)(\forall y)\, P(x,y)$ **with** w;
  specialize $(\forall y)\, P(w,y)$ **with** z
end

$$\boxed{(\exists x)(\exists y)\, P(x,y) \Rightarrow (\exists y)(\exists x)\, P(x,y)}$$

Proof:
assume $(\exists x)(\exists y)\, P(x,y)$ **in**
  pick-witness x_1 **for** $(\exists x)(\exists y)\, P(x,y)$ **in**
    pick-witness y_1 **for** $(\exists y)\, P(x_1,y)$ **in**
      begin
        ex-generalize $(\exists x)\, P(x,y_1)$ **from** x_1;
        ex-generalize $(\exists y)(\exists x)\, P(x,y)$ **from** y_1
      end

$$\boxed{(\exists x)(\forall y)\, P(x,y) \Rightarrow (\forall y)(\exists x)\, P(x,y)}$$

Proof:
assume $(\exists x)(\forall y)\, P(x,y)$ **in**
  pick-any z **in**
    pick-witness x_1 **for** $(\exists x)(\forall y)\, P(x,y)$ **in**
      begin
        specialize $(\forall y)\, P(x_1,y)$ **with** z;
        ex-generalize $(\exists x)\, P(x,z)$ **from** x_1
      end

The above derivations imply the second and third parts of the following lemma, which will come handy in our completeness proof; the first part is an easy exercise in propositional reasoning.

Lemma 6.4 (a) *If* $\beta \vdash_{\mathcal{CND}} \neg(F \Rightarrow G)$ *then* $\beta \vdash_{\mathcal{CND}} F$ *and* $\beta \vdash_{\mathcal{CND}} \neg G$.

(b) *If* $\beta \vdash_{\mathcal{CND}} (\forall x)\, \neg F$ *then* $\beta \vdash_{\mathcal{CND}} \neg(\exists x)\, F$.

(c) *If* $\beta \vdash_{\mathcal{CND}} \neg(\forall x)\, F$ *then* $\beta \vdash_{\mathcal{CND}} (\exists x)\, \neg F$.

6.4 Theory

In this section we derive some fundamental results about $\mathcal{CND}$.

Lemma 6.5 *$\sigma\,\tau\,F = \sigma \circ \tau\,F$.*

Proof: By induction on the structure of F. When F is an atom $R(t_1, \ldots, t_n)$ we have

$$\sigma\,\tau\,F = R(\overline{\sigma}(\overline{\tau}(t_1)), \ldots, \overline{\sigma}(\overline{\tau}(t_n)))$$

and

$$\sigma \circ \tau\,F = R(\overline{\sigma \circ \tau}(t_1), \ldots, \overline{\sigma \circ \tau}(t_n))$$

and the result follows from Lemma A.1. When F is a negation $\neg G$ we have $\sigma\,\tau\,F = \neg\sigma\,\tau\,G$ while $\sigma \circ \tau\,F = \neg\sigma \circ \tau\,G$, so the identity follows from the inductive hypothesis. The rest of the propositional cases are similar.

Finally, when F is of the form $(Q\,x)\,G$ for some quantifier Q, we have

$$\sigma\,\tau\,F = (Q\,x)\,\sigma[x \mapsto x]\,\tau[x \mapsto x]\,G \tag{6.7}$$

while

$$\sigma \circ \tau\,F = (Q\,x)\,(\sigma \circ \tau)[x \mapsto x]\,G. \tag{6.8}$$

By alphabetic conversion, we may assume that $x \notin \mathit{Supp}(\sigma) \cup \mathit{Supp}(\tau)$, and thus

$$\sigma[x \mapsto x] = \sigma,$$

$$\tau[x \mapsto x] = \tau,$$

and

$$(\sigma \circ \tau)[x \mapsto x] = \sigma \circ \tau$$

so the identity follows directly from 6.7 and 6.8 by the inductive hypothesis. ■

Lemma 6.6 *Let $\theta_1 = \{x \mapsto y_1\}$, $\theta_2 = \{x \mapsto y_2\}$, $x \neq y_1$. Then $\theta_2 \circ \theta_1 = \theta_1$.*

Proof: We have $\theta_1(x) = y_1$ and

$$\theta_2 \circ \theta_1(x) = \overline{\theta_2}(\theta_1(x)) = \overline{\theta_2}(y_1) = y_1 = \theta_1(x)$$

(since $y_1 \notin \mathit{Supp}(\theta_2)$ by the supposition $x \neq y_1$). For any variable $z \neq x$ we also have

$$\theta_2 \circ \theta_1(z) = \overline{\theta_2}(\theta_1(z)) = z = \theta_1(z).$$

Thus we have shown that $\theta_1(v) = \theta_2 \circ \theta_1(v)$ for all variables v. ■

Corollary 6.7 *Let* $\theta_1 = \{x \mapsto y_1\}$, $\theta_2 = \{x \mapsto y_2\}$, $x \neq y_1$. *Then* $\overline{\theta_2}(\overline{\theta_1}(t)) = \overline{\theta_1}(t)$.

Proof: By Lemma A.1, $\overline{\theta_2}(\overline{\theta_1}(t)) = \overline{\theta_2 \circ \theta_1}(t)$, thus the desired identity follows from Lemma 6.6. ∎

Corollary 6.8 *Let* $\theta_1 = \{x \mapsto y_1\}$, $\theta_2 = \{x \mapsto y_2\}$, $x \neq y_1$. *Then* $\theta_2\,\theta_1\,F = \theta_1\,F$.

Proof: Immediate from Lemma 6.6 and Lemma 6.5. ∎

Lemma 6.9 *Let* $\theta_1 = \{x \mapsto y_1\}$, $\theta_2 = \{x \mapsto y_2\}$, $x \neq y_1$. *Then* $\theta_2\,\theta_1\,D = \theta_1\,D$.

Proof: By induction on the structure of D. When D is a claim F or a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$, or **specialize** F **with** t, or

$$\textbf{ex-generalize } F \textbf{ from } t,$$

the result follows from Corollary 6.7 and Corollary 6.8. When D is of the form **assume** F **in** D' we have

$$\theta_2\,\theta_1\,D = \textbf{assume } \theta_2\,\theta_1\,F \textbf{ in } \theta_2\,\theta_1\,D'$$

while

$$\theta_1\,D = \textbf{assume } \theta_1\,F \textbf{ in } \theta_1\,D'$$

and the identity now follows from Corollary 6.8 and the inductive hypothesis. When D is of the form $D_1; D_2$, the result follows readily from the inductive hypothesis.

Next, when D is of the form **pick-any** z **in** D', we have

$$\theta_2\,\theta_1\,D = \textbf{pick-any } z \textbf{ in } \theta_2[z \mapsto z]\,\theta_1[z \mapsto z]\,D' \tag{6.9}$$

while

$$\theta_1\,D = \textbf{pick-any } z \textbf{ in } \theta_1[z \mapsto z]\,D' \tag{6.10}$$

There are two cases:

(a) $z = x$: Then $\theta_1[z \mapsto z] = \{\} = \theta_2[z \mapsto z]$, so 6.9 and 6.10 yield $\theta_2\,\theta_1\,D = \theta_1\,D$.

(b) $z \neq x$: Then $\theta_1[z \mapsto z] = \theta_1$, $\theta_2[z \mapsto z] = \theta_2$, and $\theta_2\,\theta_1\,D = \theta_1\,D$ now follows from 6.9, 6.10, and the inductive hypothesis.

Finally, suppose that D is of the form **pick-witness** z **for** F **in** D'. Then

$$\theta_2\,\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_2\,\theta_1\,F \textbf{ in } \theta_2[z \mapsto z]\,\theta_1[z \mapsto z]\,D' \tag{6.11}$$

while

$$\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_1\,F \textbf{ in } \theta_1[z \mapsto z]\,D' \tag{6.12}$$

and we again perform a case analysis:

(a) $z = x$: Then $\theta_1[z \mapsto z] = \{\} = \theta_2[z \mapsto z]$, so 6.11 and 6.12 yield

$$\theta_2\,\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_2\,\theta_1\,F \textbf{ in } D'$$

while

$$\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_1\,F \textbf{ in } D'$$

and thus the equality $\theta_2\,\theta_1\,D = \theta_1\,D$ follows from Corollary 6.8.

(b) $z \neq x$: Then $\theta_1[z \mapsto z] = \theta_1$, $\theta_2[z \mapsto z] = \theta_2$, so 6.11 and 6.12 become

$$\theta_2\,\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_2\,\theta_1\,F \textbf{ in } \theta_2\,\theta_1\,D'$$

while

$$\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_1\,F \textbf{ in } \theta_1\,D'$$

and $\theta_2\,\theta_1\,D = \theta_1\,D$ now follows from Corollary 6.8 and the inductive hypothesis.

This completes the case analysis and the inductive argument. ∎

We will call two substitutions θ_1 and θ_2 *disjoint* iff

1. $Supp(\theta_1) \cap Supp(\theta_2) = \emptyset$; and
2. $RanVar(\theta_i) \cap Supp(\theta_j) = \emptyset$ whenever $i, j \in \{1, 2\}, i \neq j$.[3]

Lemma 6.10 *If θ_1 and θ_2 are disjoint then so are $\theta_1[x \mapsto x]$ and $\theta_2[x \mapsto x]$.*

Proof: The reader will recall the following elementary fact about sets: if $S_1 \cap S_2 = \emptyset$ then $S'_1 \cap S'_2 = \emptyset$ for all $S'_1 \subseteq S_1$, $S'_2 \subseteq S_2$. Thus the result is immediate, since $Supp(\theta_i[x \mapsto x]) \subseteq Supp(\theta_i)$ and $RanVar(\theta_i[x \mapsto x]) \subseteq RanVar(\theta_i)$ for $i = 1, 2$. ∎

The following result shows that disjoint substitutions commute:

Lemma 6.11 *If θ_1, θ_2 are disjoint then $\theta_2 \circ \theta_1 = \theta_1 \circ \theta_2$.*

Proof: We will show that $L(x) = R(x)$ for all variables x, where $L(x) = \overline{\theta_2}(\theta_1(x))$ and $R(x) = \overline{\theta_1}(\theta_2(x))$. There are two possible cases:

[3] See page 392 for the definition of *RanVar*.

(a) $x \in \mathit{Supp}(\theta_2)$: Then, by the supposition of disjointness,

$$x \notin \mathit{Supp}(\theta_1) \tag{6.13}$$

and

$$\mathit{Var}(\theta_2(x)) \cap \mathit{Supp}(\theta_1) = \emptyset. \tag{6.14}$$

Therefore, from 6.13, $L(x) = \overline{\theta_2}(\theta_1(x)) = \theta_2(x)$, while from 6.14,

$$R(x) = \overline{\theta_1}(\theta_2(x)) = \theta_2(x)$$

and thus $L(x) = R(x)$.

(b) $x \notin \mathit{Supp}(\theta_2)$: Here we distinguish two subcases:

- $x \in \mathit{Supp}(\theta_1)$: Then, by disjointness,

$$\mathit{Var}(\theta_1(x)) \cap \mathit{Supp}(\theta_2) = \emptyset \tag{6.15}$$

  hence $L(x) = \overline{\theta_2}(\theta_1(x)) = \theta_1(x)$, while $R(x) = \overline{\theta_1}(\theta_2(x)) = \theta_1(x)$, thus the equality holds in this case as well.

- $x \notin \mathit{Supp}(\theta_1)$: Then $L(x) = x = R(x)$.

The proof is now complete since the considered cases are jointly exhaustive. ■

Corollary 6.12 *If θ_1 and θ_2 are disjoint then $\overline{\theta_2}(\overline{\theta_1}(t)) = \overline{\theta_1}(\overline{\theta_2}(t))$.*

Proof: By Lemma 6.11 and Lemma A.1. ■

The following is the analogue of Corollary 6.12 for formulas:

Corollary 6.13 *$\theta_2\, \theta_1\, F = \theta_1\, \theta_2\, F$ whenever θ_1, θ_2 are disjoint.*

Proof: By Lemma 6.11 and Lemma 6.5. ■

We can generalize the result to deductions as follows:

Lemma 6.14 *$\theta_2\, \theta_1\, D = \theta_1\, \theta_2\, D$ whenever θ_1, θ_2 are disjoint.*

Proof: By induction on the structure of D. When D is a claim F or a primitive deduction of the form

$$\textit{Prim-Rule}\ F_1, \ldots, F_n$$

or **specialize** F **with** t, or **ex-generalize** F **from** t, the result follows from Corollary 6.13 and Corollary 6.12. When D is of the form **assume** F **in** D', we have

$$\theta_2\,\theta_1\,D = \textbf{assume } \theta_2\,\theta_1\,F \textbf{ in } \theta_2\,\theta_1\,D'$$

while

$$\theta_1\,\theta_2\,D = \textbf{assume } \theta_1\,\theta_2\,F \textbf{ in } \theta_1\,\theta_2\,D'$$

and the desired identity follows from Corollary 6.13 and the inductive hypothesis. When D is a composite deduction $D_1; D_2$, the result follows directly from the inductive hypothesis.

Next, when D is of the form **pick-any** z **in** D', we have

$$\theta_2\,\theta_1\,D = \textbf{pick-any } z \textbf{ in } \theta_2[z \mapsto z]\,\theta_1[z \mapsto z]\,D'$$

and

$$\theta_1\,\theta_2\,D = \textbf{pick-any } z \textbf{ in } \theta_1[z \mapsto z]\,\theta_2[z \mapsto z]\,D'$$

and since $\theta_1[z \mapsto z]$ and $\theta_2[z \mapsto z]$ are disjoint (Lemma 6.10), the result follows from the inductive hypothesis. Finally, when D is of the form **pick-witness** z **for** F **in** D',

$$\theta_2\,\theta_1\,D = \textbf{pick-witness } z \textbf{ for } \theta_2\,\theta_1\,F \textbf{ in } \theta_2[z \mapsto z]\,\theta_1[z \mapsto z]\,D'$$

and

$$\theta_1\,\theta_2\,D = \textbf{pick-witness } z \textbf{ for } \theta_1\,\theta_2\,F \textbf{ in } \theta_1[z \mapsto z]\,\theta_2[z \mapsto z]\,D'$$

and the result now follows from Corollary 6.13 and the inductive hypothesis, since $\theta_1[z \mapsto z]$ and $\theta_2[z \mapsto z]$ are disjoint (again, by Lemma 6.10). ■

Lemma 6.15 *Let* $\sigma = \{x_1 \mapsto x_2\}$, $\tau = \{x_2 \mapsto x_3\}$, $\theta = \{x_1 \mapsto x_3\}$. *If* $x_2 \notin Var(t)$ *then* $\overline{\tau}(\overline{\sigma}(t)) = \overline{\theta}(t)$.

Proof: By structural induction on t. When t is a variable, it is either x_1 or something other than x_1 and x_2 (it cannot be x_2 by supposition). If it is x_1 then $\overline{\tau}(\overline{\sigma}(t)) = x_3$, while $\overline{\theta}(t) = x_3$, so the equality holds. When it is something other than x_1 and x_2 then

$$\overline{\tau}(\overline{\sigma}(t)) = t = \overline{\theta}(t)$$

so the equality holds in this case too. When t is a constant symbol the result is immediate. Finally, when t is of the form $f(t_1, \ldots, t_n)$ we have

$$\overline{\tau}(\overline{\sigma}(t)) = f(\overline{\tau}(\overline{\sigma}(t_1)), \ldots, \overline{\tau}(\overline{\sigma}(t_n)))$$

and

$$\overline{\theta}(t) = f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))$$

so the equality follows from the inductive hypothesis. ■

We immediately get:

Corollary 6.16 *If $x_2 \notin Var(t)$ then* $\overline{\{x_2 \mapsto x_1\}}(\overline{\{x_1 \mapsto x_2\}}(t)) = t$.

The following is the analogue of Lemma 6.15 for formulas:

Lemma 6.17 *Let $\sigma = \{x_1 \mapsto x_2\}$, $\tau = \{x_2 \mapsto x_3\}$, $\theta = \{x_1 \mapsto x_3\}$. If $x_2 \notin Var(F)$ then $\tau\,\sigma\,F = \theta\,F$.*

Proof: By structural induction on F. When F is an atom the result follows from Lemma 6.15. When F is a negation $\neg G$, we have $\tau\,\sigma\,F = \neg\tau\,\sigma\,G$, and, by the inductive hypothesis, $\tau\,\sigma\,G = \theta\,G$, hence

$$\tau\,\sigma\,F = \neg\theta\,G = \theta\,\neg G = \theta\,F.$$

The rest of the propositional cases are handled similarly.

Finally, suppose that F is of the form $(Q\,v)\,G$. Owing to alphabetic conversion, we may assume that

$$v \notin \{x_1, x_2, x_3\}$$

so that $\sigma[v \mapsto v] = \sigma$, $\tau[v \mapsto v] = \tau$, and $\theta[v \mapsto v] = \theta$. Accordingly,

$$\tau\,\sigma\,F = (Q\,v)\,\tau\,\sigma\,G \tag{6.16}$$

and

$$\theta\,F = (Q\,v)\,\theta\,G. \tag{6.17}$$

Since x_2 does not occur in G, the inductive hypothesis yields

$$\tau\,\sigma\,G = \theta\,G$$

hence

$$\tau\,\sigma\,F = \theta\,F$$

now follows from 6.16 and 6.17. ■

Corollary 6.18 *If $x_2 \notin Var(F)$ then $\{x_2 \mapsto x_1\}\,\{x_1 \mapsto x_2\}\,F = F$.*

The following result generalizes Lemma 6.17 to deductions:

Lemma 6.19 *Let $\sigma = \{x_1 \mapsto x_2\}$, $\tau = \{x_2 \mapsto x_3\}$, $\theta = \{x_1 \mapsto x_3\}$. If x_2 does not occur in D then $\tau\,\sigma\,D = \theta\,D$.*

Proof: By induction on the structure of D. When D is a claim F or a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$, the the result follows from Lemma 6.17. When D is of the form **specialize** F **with** t then

$$\tau\,\sigma\,D = \textbf{specialize}\ \tau\,\sigma\,F\ \textbf{with}\ \overline{\tau}(\overline{\sigma}(t))$$

while

$$\theta\,D = \textbf{specialize}\ \theta\,F\ \textbf{with}\ \overline{\theta}(t)$$

and the result follows from Lemma 6.15 and Lemma 6.17. When D is of the form

$$\textbf{ex-generalize}\ F\ \textbf{from}\ t$$

the reasoning is similar. When D is of the form **assume** F **in** D' we have

$$\tau\,\sigma\,D = \textbf{assume}\ \tau\,\sigma\,F\ \textbf{in}\ \tau\,\sigma\,D'$$

while

$$\theta\,D = \textbf{assume}\ \theta\,F\ \textbf{in}\ \theta\,D'$$

and the desired identity now follows from Lemma 6.17 and the inductive hypothesis. Next, when D is of the form **pick-any** z **in** D', we have

$$\tau\,\sigma\,D = \tau\,\textbf{pick-any}\ z\ \textbf{in}\ \sigma[z \mapsto z]\,D'. \tag{6.18}$$

We distinguish two cases:

(a) $z = x_1$: Then $\sigma[z \mapsto z] = \{\}$, hence, from 6.18,

$$\begin{aligned}\tau\,\sigma\,D &= \tau\,\textbf{pick-any}\ z\ \textbf{in}\ D'\\ &= \textbf{pick-any}\ z\ \textbf{in}\ \tau[z \mapsto z]\,D'\\ &= \textbf{pick-any}\ z\ \textbf{in}\ D'\end{aligned}$$

where the last identity holds because x_2 does not occur in D, and thus

$$\tau[z \mapsto z]\,D' = \tau\,D' = D'.$$

Furthermore,

$$\theta\,D = \textbf{pick-any}\ z\ \textbf{in}\ \theta[z \mapsto z]\,D'$$

and since $z = x_1$, $\theta[z \mapsto z]$ is the empty substitution $\{\}$, hence

$$\theta\,D = \textbf{pick-any}\ z\ \textbf{in}\ D'$$

and we thus have $\tau\,\sigma\,D = \theta\,D$.

(b) $z \neq x_1$: Then $\sigma[z \mapsto z] = \sigma$, hence, from 6.18,

$$\begin{aligned} \tau\,\sigma\,D &= \tau\,\textbf{pick-any}\ z\ \textbf{in}\ \sigma\,D' \\ &= \textbf{pick-any}\ z\ \textbf{in}\ \tau[z \mapsto z]\,\sigma\,D'. \end{aligned}$$

But x_2 does not occur in D, hence $\tau[z \mapsto z] = \tau$, and therefore

$$\tau\,\sigma\,D = \textbf{pick-any}\ z\ \textbf{in}\ \tau\,\sigma\,D'.$$

Likewise, $z \neq x_1$ entails $\theta[z \mapsto z] = \theta$, thus

$$\theta\,D = \textbf{pick-any}\ z\ \textbf{in}\ \theta\,D'$$

and the equality $\tau\,\sigma\,D = \theta\,D$ now follows from the inductive hypothesis.

Finally, suppose that D is of the form **pick-witness** z **for** F **in** D'. Then

$$\tau\,\sigma\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \tau\,\sigma\,F\ \textbf{in}\ \tau[z \mapsto z]\,\sigma[z \mapsto z]\,D' \tag{6.19}$$

while

$$\theta\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \theta\,F\ \textbf{in}\ \theta[z \mapsto z]\,D'. \tag{6.20}$$

We again distinguish two cases:

(a) $z = x_1$: Then $\sigma[z \mapsto z] = \theta[z \mapsto z] = \{\}$, so 6.19 and 6.20 become

$$\tau\,\sigma\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \tau\,\sigma\,F\ \textbf{in}\ \tau[z \mapsto z]\,D' \tag{6.21}$$

and

$$\theta\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \theta\,F\ \textbf{in}\ D'. \tag{6.22}$$

But x_2 does not occur in D, hence $\tau[z \mapsto z]\,D' = \tau\,D' = D'$, and the result now follows from 6.21, 6.22, and Lemma 6.17.

(b) $z \neq x_1$: Then $\sigma[z \mapsto z] = \sigma$, $\tau[z \mapsto z] = \tau$ (since x_2 does not occur in D), and $\theta[z \mapsto z] = \theta$, so 6.19 and 6.20 become

$$\tau\,\sigma\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \tau\,\sigma\,F\ \textbf{in}\ \tau\,\sigma\,D'$$

and

$$\theta\,D = \textbf{pick-witness}\ z\ \textbf{for}\ \theta\,F\ \textbf{in}\ \theta\,D'.$$

The result now follows from Lemma 6.17 and the inductive hypothesis.

This completes the case analysis and the inductive argument. ■

Corollary 6.20 *If x_2 does not occur in D then $\{x_2 \mapsto x_1\}\,\{x_1 \mapsto x_2\}\,D = D$.*

Lemma 6.21 *If $x \notin \mathit{Var}(t) \cup \mathit{RanVar}(\theta)$ then $x \notin \mathit{Var}(\overline{\theta}(t))$.*

Proof: By induction on t. When t is a variable z we must have $z \neq x$ (since $x \notin \mathit{Var}(t)$), and there are two possibilities:

(a) $z \in \mathit{Supp}(\theta)$: Then
$$x \notin \mathit{Var}(\overline{\theta}(t)) = \mathit{Var}(\theta(z))$$
by the supposition $x \notin \mathit{RanVar}(\theta)$.

(b) $z \notin \mathit{Supp}(\theta)$: Then
$$\mathit{Var}(\overline{\theta}(t)) = \mathit{Var}(\theta(z)) = \{z\}$$
and the result follows because $x \neq z$.

When t is a constant symbol the result is trivial; and for an application $f(t_1, \ldots, t_n)$ it follows readily from the inductive hypothesis. ■

Lemma 6.22 *If $x \notin FV(F) \cup \mathit{RanVar}(\theta)$ then $x \notin \mathit{Var}(\theta\, F)$.*

Proof: By induction on F. When F is an atom the result follows from Lemma 6.21. When F is a negation $\neg G$ we have $x \notin FV(G)$ and $\theta\, F = \neg\theta\, G$, thus the inductive hypothesis yields
$$x \notin \mathit{Var}(\theta\, G) = \mathit{Var}(\neg\theta\, G) = \mathit{Var}(\theta\, F).$$
The rest of the propositional cases are similar. Finally, suppose that D is of the form $(Q\, z)\, G$ for some quantifier Q. By alphabetic conversion, we may assume that $z \neq x$, so that $x \notin FV(G)$ (since $FV(G) \subseteq FV(F) \cup \{z\}$ and $x \notin FV(F)$). Further, $x \notin \mathit{RanVar}(\theta[z \mapsto z])$, thus the inductive hypothesis yields
$$x \notin \mathit{Var}(\theta[z \mapsto z]\, G).$$
Since $x \neq z$,
$$x \notin \mathit{Var}((Q\, z)\, \theta[z \mapsto z]\, G) = \mathit{Var}(\theta\, (Q\, z)\, G) = \mathit{Var}(\theta\, F)$$
and the induction is complete. ■

Lemma 6.23 *If x does not occur in D or in $RanVar(\theta)$ then x does not occur in $\theta\, D$.*

Proof: By induction on the structure of D. When D is a claim or a primitive deduction the result follows from Lemma 6.21 and Lemma 6.22. When D is a hypothetical deduction **assume** F **in** D', the result follows from Lemma 6.22 and the inductive hypothesis. In the case of a composite deduction $D_1; D_2$ the result is immediate from the inductive hypothesis.

Next, suppose that D is of the form **pick-any** z **in** D'. Then

$$\theta\, D = \textbf{pick-any } z \textbf{ in } \theta[z \mapsto z]\, D'.$$

But x does not occur in D' or in $RanVar(\theta[z \mapsto z])$, so the inductive hypothesis entails that x does not occur in $\theta[z \mapsto z]\, D'$, and since $z \neq x$, x does not occur in

$$\textbf{pick-any } z \textbf{ in } \theta[z \mapsto z]\, D' = \theta\, D.$$

The case when D is of the form **pick-witness** z **for** F **in** D' is similar, except that we also need to invoke Lemma 6.22 to show that $x \notin Var(\theta\, F)$. ∎

Lemma 6.24 *If $x \notin Var(t)$ then $\overline{\theta}(t) = \overline{\theta[x \mapsto x]}(t)$.*

Proof: By induction on t. When t is a variable y, we must have $y \neq x$, so $\overline{\theta}(t) = \theta(y)$ while

$$\overline{\theta[x \mapsto x]}(t) = \theta[x \mapsto x](y) = \theta(y)$$

and the equality holds. When t is a constant symbol the result is immediate; and when it is an application, the result follows readily from the inductive hypothesis. ∎

Lemma 6.25 *If $x \notin FV(F)$ then $\theta\, F = \theta[x \mapsto x]\, F$.*

Proof: By induction on F. When F is an atom the result follows from Lemma 6.24. Propositional cases are handled by straightforward applications of the inductive hypothesis. Finally, when F is a quantified formula $(Q\, y)\, G$ we may assume (by renaming) that $y \notin Supp(\theta) \cup \{x\}$, so that $\theta\, F = (Q\, y)\, \theta\, G$ and $x \notin FV(G)$. Inductively,

$$\theta\, G = \theta[x \mapsto x]\, G$$

thus

$$\theta\, F = (Q\, y)\, \theta[x \mapsto x]\, G. \tag{6.23}$$

The supposition $y \notin Supp(\theta)$ entails $\theta[x \mapsto x] = \theta[x \mapsto x][y \mapsto y]$, thus 6.23 becomes

$$\theta\, F = (Q\, y)\, \theta[x \mapsto x][y \mapsto y]\, G = \theta[x \mapsto x]\, (Q\, y)\, G = \theta[x \mapsto x]\, F$$

and the induction is complete. ∎

Lemma 6.26 *If $x \in Supp(\theta)$ and $x \notin RanVar(\theta)$ then $x \notin Var(\overline{\theta}(t))$.*

Proof: By induction on the structure of t. When t is a variable there are two cases:

(a) $t = x$: Then the suppositions $x \notin RanVar(\theta)$ and $x \in Supp(\theta)$ imply that x does not occur in $Var(\theta(x)) = Var(\overline{\theta}(t))$.

(b) $t \neq x$: Here we distinguish two subcases:

1. $t \in Supp(\theta)$: Then $x \notin RanVar(\theta)$ implies $x \notin Var(\overline{\theta}(t))$.
2. $t \notin Supp(\theta)$: Then $\overline{\theta}(t) = t$, and $x \notin Var(\overline{\theta}(t))$ now follows from the assumption $x \neq t$.

When t is a constant symbol the result is trivial; and when it is an application $f(t_1, \ldots, t_n)$, it follows from a straightforward application of the inductive hypothesis. ■

Lemma 6.27 *If $x \in Supp(\theta)$ and $x \notin RanVar(\theta)$ then $x \notin Var(\theta\, F)$.*

Proof: By induction on the structure of F. When F is an atom the result follows from Lemma 6.26. When F is a negation $\neg G$ we have $\theta\, F = \neg\theta\, G$, and, inductively, we have $x \notin Var(\theta\, G)$, hence $x \notin Var(\neg\theta\, G) = Var(\theta\, F)$. The rest of the propositional cases are similar.

Finally, suppose that F is of the form $(Q\, z)\, G$ for some quantifier Q. Then, by alphabetic conversion, we may assume $z \notin Supp(\theta)$, so that $\theta[z \mapsto z] = \theta$ and $\theta\,(Q\, z)\, G = (Q\, z)\, \theta\, G$. The result now follows from the inductive hypothesis, since $z \notin x$. ■

Lemma 6.28 *If $x \in Supp(\theta)$ and $x \notin RanVar(\theta)$ then $\sigma\, \theta\, D = \sigma[x \mapsto x]\, \theta\, D$.*

Proof: By induction on the structure of D. When D is a claim F, Lemma 6.27 implies $x \notin Var(\theta\, F)$, hence by Lemma 6.25,

$$\sigma\, \theta\, D = \sigma\, (\theta\, F) = \sigma[x \mapsto x]\, \theta\, F = \sigma[x \mapsto x]\, \theta\, D.$$

When D is a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$,

$$\sigma\, \theta\, D = \textit{Prim-Rule}\ \sigma\, \theta\, F_1, \sigma\, \theta\, F_n$$

and again by Lemma 6.27, $x \notin Var(\theta\, F_i)$, hence Lemma 6.25 entails

$$\sigma\, \theta\, F_i = \sigma[x \mapsto x]\, \theta\, F_i.$$

Accordingly,

$$\begin{aligned}\sigma\,\theta\,D &= \textit{Prim-Rule}\ \sigma[x\mapsto x]\,\theta\,F_1,\ldots,\sigma[x\mapsto x]\,\theta\,F_n\\ &= \sigma[x\mapsto x]\,\theta\,\textit{Prim-Rule}\ F_1,\ldots,F_n\\ &= \sigma[x\mapsto x]\,\theta\,D.\end{aligned}$$

When D is of the form **specialize** F **with** t, we have

$$\sigma\,\theta\,D = \textbf{specialize}\ \sigma\,\theta\,F\ \textbf{with}\ \overline{\sigma}(\overline{\theta}(t)). \tag{6.24}$$

Reasoning as above, $\sigma\,\theta\,F = \sigma[x\mapsto x]\,\theta\,F$, while by Lemma 6.26 and Lemma 6.24 we obtain

$$\overline{\sigma}(\overline{\theta}(t)) = \overline{\sigma[x\mapsto x]}(\overline{\theta}(t)).$$

Thus 6.24 becomes

$$\begin{aligned}\sigma\,\theta\,D &= \textbf{specialize}\ \sigma[x\mapsto x]\,\theta\,F\ \textbf{with}\ \overline{\sigma[x\mapsto x]}(\overline{\theta}(t))\\ &= \sigma[x\mapsto x]\,\theta\ \textbf{specialize}\ F\ \textbf{with}\ t\\ &= \sigma[x\mapsto x]\,\theta\,D.\end{aligned}$$

Similar reasoning applies when D is of the form **ex-generalize** F **from** t.

When D is a hypothetical deduction **assume** F **in** D',

$$\sigma\,\theta\,D = \textbf{assume}\ \sigma\,\theta\,F\ \textbf{in}\ \sigma\,\theta\,D'. \tag{6.25}$$

Again by Lemma 6.27 and Lemma 6.25 we get

$$\sigma\,\theta\,F = \sigma[x\mapsto x]\,\theta\,F$$

while the inductive hypothesis gives $\sigma\,\theta\,D' = \sigma[x\mapsto x]\,\theta\,D'$, so 6.25 becomes

$$\begin{aligned}\sigma\,\theta\,D &= \textbf{assume}\ \sigma[x\mapsto x]\,\theta\,F\ \textbf{in}\ \sigma[x\mapsto x]\,\theta\,D'\\ &= \sigma[x\mapsto x]\,\theta\ \textbf{assume}\ F\ \textbf{in}\ D'\\ &= \sigma[x\mapsto x]\,\theta\,D.\end{aligned}$$

When D is a composition $D_1; D_2$ the result follows directly from the inductive hypothesis.

Next, when D is of the form **pick-any** z **in** D', we have

$$\sigma\,\theta\,D = \textbf{pick-any}\ z\ \textbf{in}\ \sigma[z\mapsto z]\,\theta[z\mapsto z]\,D'. \tag{6.26}$$

There are two possibilities:

(a) $x = z$: Then

$$\begin{array}{rcll} \sigma[x \mapsto x]\,\theta\, D & = & \textbf{pick-any } z \textbf{ in } \sigma[x \mapsto x][x \mapsto x]\,\theta[x \mapsto x]\, D' & \\ & = & \textbf{pick-any } z \textbf{ in } \sigma[x \mapsto x]\,\theta[x \mapsto x]\, D' & \text{(from 6.26, since } x = z) \\ & = & \sigma\,\theta\, D. & \end{array}$$

(b) $x \neq z$: Then $x \in \mathit{Supp}(\theta[z \mapsto z])$ (since $x \in \mathit{Supp}(\theta)$ and $x \neq z$), and

$$x \notin \mathit{RanVar}(\theta[z \mapsto z])$$

so the inductive hypothesis yields

$$\begin{array}{rcl} \sigma[z \mapsto z]\,\theta[z \mapsto z]\, D' & = & \sigma[z \mapsto z][x \mapsto x]\,\theta[z \mapsto z]\, D' \\ & = & \sigma[x \mapsto x][z \mapsto z]\,\theta[z \mapsto z]\, D'. \end{array}$$

Thus 6.26 becomes

$$\begin{array}{rcl} \sigma\,\theta\, D & = & \textbf{pick-any } z \textbf{ in } \sigma[x \mapsto x][z \mapsto z]\,\theta[z \mapsto z]\, D' \\ & = & \sigma[x \mapsto x]\,\theta\ \ \textbf{pick-any } z \textbf{ in } D' \\ & = & \sigma[x \mapsto x]\,\theta\, D. \end{array}$$

Finally, when D is of the form **pick-witness** z **for** F **in** D', we have

$$\sigma\,\theta\, D = \textbf{pick-witness } z \textbf{ for } \sigma\,\theta\, F \textbf{ in } \sigma[z \mapsto z]\,\theta[z \mapsto z]\, D'. \tag{6.27}$$

By Lemma 6.27 and Lemma 6.25 we get

$$\sigma\,\theta\, F = \sigma[x \mapsto x]\,\theta\, F \tag{6.28}$$

and we again distinguish two cases:

(a) $z = x$: Then

$$\begin{array}{rcll} \sigma[x \mapsto x]\,\theta\, D & = & \textbf{pick-witness } z \textbf{ for } \sigma[x \mapsto x]\,\theta\, F \textbf{ in } \sigma[z \mapsto z][z \mapsto z]\,\theta[z \mapsto z]\, D' & \\ & = & \textbf{pick-witness } z \textbf{ for } \sigma[x \mapsto x]\,\theta\, F \textbf{ in } \sigma[z \mapsto z]\,\theta[z \mapsto z]\, D' & \text{(from 6.28)} \\ & = & \textbf{pick-witness } z \textbf{ for } \sigma\,\theta\, F \textbf{ in } \sigma[z \mapsto z]\,\theta[z \mapsto z]\, D' & \text{(from 6.27)} \\ & = & \sigma\,\theta\, D. & \end{array}$$

(b) $z \neq x$: Then $x \in \mathit{Supp}(\theta[z \mapsto z])$, $x \notin \mathit{RanVar}(\theta[z \mapsto z])$, so the inductive hypothesis gives

$$\begin{array}{rcl} \sigma[z \mapsto z]\,\theta[z \mapsto z]\, D' & = & \sigma[z \mapsto z][x \mapsto x]\,\theta[z \mapsto z]\, D' \\ & = & \sigma[x \mapsto x][z \mapsto z]\,\theta[z \mapsto z]\, D' \end{array}$$

and thus 6.27 becomes

$$\begin{array}{rcll} \sigma\,\theta\,D & = & \textbf{pick-witness}\ z\ \textbf{for}\ \sigma\,\theta\,F\ \textbf{in}\ \sigma[x\mapsto x][z\mapsto z]\,\theta[z\mapsto z]\,D' & \text{(from 6.28)}\\ & = & \textbf{pick-witness}\ z\ \textbf{for}\ \sigma[x\mapsto x]\,\theta\,F\ \textbf{in}\ \sigma[x\mapsto x][z\mapsto z]\,\theta[z\mapsto z]\,D' & \\ & = & \sigma[x\mapsto x]\,\theta\ \textbf{pick-witness}\ z\ \textbf{for}\ F\ \textbf{in}\ D' & \\ & = & \sigma[x\mapsto x]\,\theta\,D. & \end{array}$$

This completes the case analysis and the induction. ■

Corollary 6.29 *If $x \notin Var(t)$ then $\theta\,\{x \mapsto t\}\,D = \theta[x \mapsto x]\,\{x \mapsto t\}\,D$.*

Lemma 6.30 *If $x \notin Supp(\theta) \cup RanVar(\theta)$ then $\{x \mapsto \overline{\theta}(t)\} \circ \theta = \theta \circ \{x \mapsto t\}$.*

Proof: We will prove that $\sigma(z) = \tau(z)$ for all variables z, where

$$\sigma = \{x \mapsto \overline{\theta}(t)\} \circ \theta$$

and

$$\tau = \theta \circ \{x \mapsto t\}.$$

We distinguish two cases:

(a) $z = x$: Then $\theta(z) = z$ (since $z = x \notin Supp(\theta)$), thus

$$\sigma(z) = \overline{\theta}(t) = \tau(z).$$

(b) $z \neq x$: Here we distinguish two subcases:

1. $z \in Supp(\theta)$: Then $x \notin Var(\theta(z))$ (since $x \notin RanVar(\theta)$), hence

$$\sigma(z) = \theta(z) = \tau(z).$$

2. $z \notin Supp(\theta)$: Then, since $z \neq x$, $\sigma(z) = z = \tau(z)$.

This completes the case analysis and the proof. ■

Corollary 6.31 $\overline{\{x \mapsto \overline{\theta}(t)\}} \cdot \overline{\theta} = \overline{\theta} \cdot \overline{\{x \mapsto t\}}$ *whenever $x \notin Supp(\theta) \cup RanVar(\theta)$.*

Proof: Immediate from Lemma 6.30 and Lemma A.1. ■

Corollary 6.32 *$\theta\,\{x \mapsto t\}\,F = \{x \mapsto \overline{\theta}(t)\}\,\theta\,F$ whenever $x \notin Supp(\theta) \cup RanVar(\theta)$.*

Proof: By Lemma 6.5 and Lemma 6.30. ■

Theorem 6.33 *If $\beta \vdash D \leadsto F$ and θ is safe for D then $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.*

Proof: By structural induction on D. When D is a claim, the supposition $\beta \vdash D \leadsto F$ means that D is the formula F and that $\beta = \beta' \cup \{F\}$ for some β' (since β must contain F for the judgment $\beta \vdash F \leadsto F$ to be derivable). Therefore, by $[R_3]$,

$$\theta\,\beta = \theta\,\beta' \cup \{\theta\,F\} \vdash \theta\,F \leadsto \theta\,F$$

which is to say $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

When D is a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$, the result follows by a case analysis of *Prim-Rule*. We illustrate with **both** and **left-either**, and leave the rest as an easy exercise. When D is of the form **both** F_1, F_2, we must have $F = F_1 \wedge F_2$ and $\beta = \beta' \cup \{F_1, F_2\}$ for some β'. Accordingly, the evaluation semantics give

$$\theta\,\beta' \cup \{\theta\,F_1, \theta\,F_2\} \vdash \textbf{both}\ \theta\,F_1, \theta\,F_2 \leadsto \theta\,F_1 \wedge \theta\,F_2$$

i.e., $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$. When D is of the form **left-either** F_1, F_2, we have $F = F_1 \vee F_2$ and $\beta = \beta' \cup \{F_1\}$. Now the semantics give

$$\theta\,\beta' \cup \{\theta\,F_1\} \vdash \textbf{left-either}\ \theta\,F_1, \theta\,F_2 \leadsto \theta\,F_1 \vee \theta\,F_2$$

i.e., $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

Next, suppose that D is of the form **specialize** $(\forall\,x)\,G$ **with** t, so that

$$F = \{x \mapsto t\}G$$

and $\beta = \beta' \cup \{(\forall\,x)\,G\}$ for some β'. Without loss of generality, we may assume

$$x \notin \mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta) \tag{6.29}$$

(we can always α-rename $(\forall\,x)\,G$ to ensure this). Now by the semantics of **specialize** we have

$$\theta\,\beta' \cup \{(\forall\,x)\,\theta\,G\} \vdash \textbf{specialize}\ (\forall\,x)\,\theta\,G\ \textbf{with}\ \overline{\theta}(t) \leadsto \{x \mapsto \overline{\theta}(t)\}\,\theta\,G. \tag{6.30}$$

But, by 6.29, $\theta[x \mapsto x] = \theta$, thus $(\forall\,x)\,\theta\,G = \theta\,(\forall\,x)\,G$; while, by Corollary 6.32,

$$\{x \mapsto \overline{\theta}(t)\}\,\theta\,G = \theta\,\{x \mapsto t\}\,G.$$

Accordingly, 6.30 becomes

$$\theta\,\beta' \cup \{\theta\,(\forall\,x)\,G\} \vdash \textbf{specialize}\ \theta\,(\forall\,x)\,G\ \textbf{with}\ \overline{\theta}(t) \leadsto \theta\,\{x \mapsto t\}\,G$$

which is to say $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

If D is of the form **ex-generalize** $(\exists\,x)\,G$ **from** t then $F = (\exists\,x)\,G$ and

$$\beta = \beta' \cup \{\{x \mapsto t\}\,G\}$$

where we may again assume

$$x \notin \mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta). \tag{6.31}$$

From $[R_7]$ we get

$$\theta\,\beta' \cup \{\{x \mapsto \overline{\theta}(t)\}\,\theta\,G\} \vdash \textbf{ex-generalize } (\exists\,x)\,\theta\,G \textbf{ from } \overline{\theta}(t) \leadsto (\exists\,x)\,\theta\,G. \tag{6.32}$$

But, from 6.31, $\theta[x \mapsto x] = \theta$, hence $\theta\,(\exists\,x)\,G = (\exists\,x)\,\theta\,G$; while, by Corollary 6.32, $\{x \mapsto \overline{\theta}(t)\}\,\theta\,G = \theta\,\{x \mapsto t\}\,G$. Thus 6.32 becomes

$$\theta\,\beta' \cup \{\theta\,\{x \mapsto t\}\,G\} \vdash \textbf{ex-generalize } \theta\,(\exists\,x)\,G \textbf{ from } \overline{\theta}(t) \leadsto \theta\,(\exists\,x)\,G$$

i.e., $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

When D is a hypothetical deduction **assume** H **in** D', we have $F = H \Rightarrow G$ and $\beta \cup \{H\} \vdash D' \leadsto G$. Inductively, the latter gives

$$\theta\,\beta \cup \{\theta\,H\} \vdash \theta\,D' \leadsto \theta\,G$$

so, by $[R_4]$,

$$\theta\,\beta \vdash \textbf{assume } \theta\,H \textbf{ in } \theta\,D' \leadsto \theta\,H \Rightarrow \theta\,G$$

i.e., $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

When D is a composition $D_1; D_2$, we must have $\beta \vdash D_1 \leadsto G$ and

$$\beta \cup \{G\} \vdash D_2 \leadsto F.$$

Inductively, $\theta\,\beta \vdash \theta\,D_1 \leadsto \theta\,G$ and $\theta\,\beta \cup \{\theta\,G\} \vdash \theta\,D_2 \leadsto \theta\,F$ hence, by $[R_5]$,

$$\theta\,\beta \vdash \theta\,D_1; \theta\,D_2 \leadsto \theta\,F$$

which is to say $\theta\,\beta \vdash \theta\,D \leadsto \theta\,F$.

When D is of the form **pick-any** x **in** D', the supposition $\beta \vdash D \leadsto F$ means that

$$\beta \vdash \{x \mapsto z\}\,D' \leadsto G \tag{6.33}$$

and $F = (\forall\,z)\,G$, for some z that does not occur in β or in D'. Let w be a variable that does not occur in β, D', G, or in $\mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta) \cup \{z, x\}$ (such a w must exist since there are infinitely many variables). From 6.33 and the inductive hypothesis we get

$$\{z \mapsto w\}\,\beta \vdash \{z \mapsto w\}\,\{x \mapsto z\}\,D' \rightsquigarrow \{z \mapsto w\}\,G. \tag{6.34}$$

Since z does not occur in β, $\{z \mapsto w\}\,\beta = \beta$; while, since z does not occur in D', Lemma 6.19 yields $\{z \mapsto w\}\,\{x \mapsto z\}\,D' = \{x \mapsto w\}\,D'$. Thus 6.34 becomes

$$\beta \vdash \{x \mapsto w\}\,D' \rightsquigarrow \{z \mapsto w\}\,G. \tag{6.35}$$

From 6.35 and the inductive hypothesis we get

$$\theta\,\beta \vdash \theta\,\{x \mapsto w\}\,D' \rightsquigarrow \theta\,\{z \mapsto w\}\,G. \tag{6.36}$$

From Corollary 6.29, $\theta\,\{x \mapsto w\}\,D' = \theta[x \mapsto x]\,\{x \mapsto w\}\,D'$, thus 6.36 becomes

$$\theta\,\beta \vdash \theta[x \mapsto x]\,\{x \mapsto w\}\,D' \rightsquigarrow \theta\,\{z \mapsto w\}\,G. \tag{6.37}$$

Now $x \notin RanVar(\theta)$ (since θ is safe for D), and thus from our assumptions about w it follows that $\theta[x \mapsto x]$ and $\{x \mapsto w\}$ are disjoint. Therefore, by virtue of Lemma 6.14, 6.37 becomes

$$\theta\,\beta \vdash \{x \mapsto w\}\,\theta[x \mapsto x]\,D' \rightsquigarrow \theta\,\{z \mapsto w\}\,G. \tag{6.38}$$

On the basis of our suppositions about w, Lemma 6.22 and Lemma 6.23 entail that w does not occur in $\theta\,\beta$ or in $\theta[x \mapsto x]\,D'$. Accordingly, $[R_8]$ and 6.38 yield

$$\theta\,\beta \vdash \textbf{pick-any}\ x\ \textbf{in}\ \theta[x \mapsto x]\,D' \rightsquigarrow (\forall\,w)\,\theta\,\{z \mapsto w\}\,G. \tag{6.39}$$

But $\theta = \theta[w \mapsto w]$, hence 6.39 gives

$$\theta\,\beta \vdash \textbf{pick-any}\ x\ \textbf{in}\ \theta[x \mapsto x]\,D' \rightsquigarrow (\forall\,w)\,\theta[w \mapsto w]\,\{z \mapsto w\}\,G \tag{6.40}$$

and since

$$(\forall\,w)\,\theta[w \mapsto w]\,\{z \mapsto w\}\,G = \theta\,(\forall\,w)\,\{z \mapsto w\}\,G$$

and

$$\textbf{pick-any}\ x\ \textbf{in}\ \theta[x \mapsto x]\,D' = \theta\,\textbf{pick-any}\ x\ \textbf{in}\ D',$$

6.40 becomes

$$\theta\,\beta \vdash \theta\,\textbf{pick-any}\ x\ \textbf{in}\ D' \rightsquigarrow \theta\,(\forall\,w)\,\{z \mapsto w\}\,G. \tag{6.41}$$

But $(\forall\,w)\,\{z \mapsto w\}\,G$ is alphabetically equivalent to $F = (\forall\,z)\,G$ (by our choice of w and Lemma 6.1), thus 6.41 becomes $\theta\,\beta \vdash \theta\,D \rightsquigarrow \theta\,F$.

Finally, suppose that D is of the form **pick-witness** x **for** $(\exists\,y)\,H$ **in** D'. Then $\beta \vdash D \rightsquigarrow F$ means that

$$\beta = \beta' \cup \{(\exists\,y)\,H\}, \tag{6.42}$$

$$\beta \cup \{\{y \mapsto z\}\, H\} \vdash \{x \mapsto z\}\, D' \leadsto F \tag{6.43}$$

and

$$z \notin FV(F) \tag{6.44}$$

for some z that does not occur in β or in D'.

Now let w be a variable that does not occur in β, D', F, or in

$$Supp(\theta) \cup RanVar(\theta) \cup \{z\}.$$

From 6.43 and the inductive hypothesis we get

$$\{z \mapsto w\}\, \beta \cup \{\{z \mapsto w\}\, \{y \mapsto z\}\, H\} \vdash \{z \mapsto w\}\, \{x \mapsto z\}\, D' \leadsto \{z \mapsto w\}\, F. \tag{6.45}$$

But z does not occur in β, hence $\{z \mapsto w\}\, \beta = \beta$. Moreover, z does not occur in H or in D', therefore, by Lemma 6.17 and Lemma 6.19, 6.45 becomes

$$\beta \cup \{\{y \mapsto w\}\, H\} \vdash \{x \mapsto w\}\, D' \leadsto \{z \mapsto w\}\, F. \tag{6.46}$$

From 6.44, $\{z \mapsto w\}\, F = F$, thus 6.46 becomes

$$\beta \cup \{\{y \mapsto w\}\, H\} \vdash \{x \mapsto w\}\, D' \leadsto F. \tag{6.47}$$

Now 6.47 and the inductive hypothesis give

$$\theta\, \beta \cup \{\theta\, \{y \mapsto w\}\, H\} \vdash \theta\, \{x \mapsto w\}\, D' \leadsto \theta\, F. \tag{6.48}$$

Without loss of generality, we may assume $y \notin Supp(\theta) \cup RanVar(\theta)$ (we can always rename $(\exists\, y)\, H$ to ensure this), thus θ and $\{y \mapsto w\}$ are disjoint, and Corollary 6.13 yields

$$\theta\, \{y \mapsto w\}\, H = \{y \mapsto w\}\, \theta\, H.$$

Furthermore, Corollary 6.29 gives

$$\theta\, \{x \mapsto w\} D' = \theta[x \mapsto x]\, \{x \mapsto w\}\, D'$$

and thus 6.48 becomes

$$\theta\, \beta \cup \{\{y \mapsto w\}\, \theta\, H\} \vdash \theta[x \mapsto x]\, \{x \mapsto w\}\, D' \leadsto \theta\, F. \tag{6.49}$$

Owing to the supposition that θ is safe for D and our assumptions about w, $\theta[x \mapsto x]$ and $\{x \mapsto w\}$ are disjoint, thus Lemma 6.14 gives

$$\theta[x \mapsto x]\, \{x \mapsto w\}\, D' = \{x \mapsto w\}\, \theta[x \mapsto x]\, D'$$

and 6.49 becomes

$$\theta\,\beta \cup \{\{y \mapsto w\}\,\theta\,H\} \vdash \{x \mapsto w\}\,\theta[x \mapsto x]\,D' \rightsquigarrow \theta\,F. \tag{6.50}$$

In addition, $\theta\,(\exists\,y)\,H = (\exists\,y)\,\theta\,H$ (since $y \notin \mathit{Supp}(\theta)$), so 6.50 and 6.42 give

$$\theta\,\beta' \cup \{(\exists\,y)\,\theta\,H, \{y \mapsto w\}\,\theta\,H\} \vdash \{x \mapsto w\}\,\theta[x \mapsto x]\,D' \rightsquigarrow \theta\,F. \tag{6.51}$$

Owing to our choice of w, Lemma 6.22 and Lemma 6.23 entail that w does not occur in $\theta\,\beta$ or in $\theta\,F$, or in $\theta[x \mapsto x]\,D'$. Consequently, $[R_9]$ and 6.51 yield

$$\theta\,\beta' \cup \{(\exists\,y)\,\theta\,H\} \vdash \textbf{pick-witness } x \textbf{ for } (\exists\,y)\,\theta\,H \textbf{ in } \theta[x \mapsto x]\,D' \rightsquigarrow \theta\,F$$

which is to say

$$\theta\,\beta \vdash \theta\,\textbf{pick-witness } x \textbf{ for } (\exists\,y)\,H \textbf{ in } D' \rightsquigarrow \theta\,F$$

i.e., $\theta\,\beta \vdash \theta\,D \rightsquigarrow \theta\,F$. ■

Corollary 6.34 *If $\beta \vdash \{x \mapsto y\}\,D \rightsquigarrow F$ for some y that does not occur in β or in D then $\beta \vdash \{x \mapsto z\}\,D \rightsquigarrow \{y \mapsto z\}\,F$ for any z that does not occur in β or in D.*

Proof: Since z does not occur in D, $\{y \mapsto z\}$ is safe for $\{x \mapsto y\}\,D$. Thus the supposition $\beta \vdash \{x \mapsto y\}\,D \rightsquigarrow F$ in tandem with Theorem 6.33 gives

$$\{y \mapsto z\}\,\beta \vdash \{y \mapsto z\}\,\{x \mapsto y\}\,D \rightsquigarrow \{y \mapsto z\}\,F. \tag{6.52}$$

But y does not occur in β, and thus $\{y \mapsto z\}\,\beta = \beta$; while, by Lemma 6.19 and the supposition that y that does not occur in D,

$$\{y \mapsto z\}\,\{x \mapsto y\}\,D = \{x \mapsto z\}\,D.$$

Therefore, 6.52 becomes $\beta \vdash \{x \mapsto z\}\,D \rightsquigarrow \{y \mapsto z\}\,F$. ■

Corollary 6.35 *If $\beta \vdash D \rightsquigarrow F$ and $x \notin FV(\beta)$ then $\beta \vdash \{x \mapsto y\}\,D \rightsquigarrow \{x \mapsto y\}\,F$ for any y that does not occur in D.*

Proof: If y does not occur in D then $\{x \mapsto y\}$ is safe for D, hence, in tandem with $\beta \vdash D \rightsquigarrow F$, Theorem 6.33 entails

$$\{x \mapsto y\}\,\beta \vdash \{x \mapsto y\}\,D \rightsquigarrow \{x \mapsto y\}\,F.$$

But because $x \notin FV(\beta)$, $\{x \mapsto y\}\,\beta = \beta$, hence $\beta \vdash \{x \mapsto y\}\,D \rightsquigarrow \{x \mapsto y\}\,F$. ■

In many natural-deduction systems, universal generalizations are introduced by an inference rule that derives $(\forall\,x)\,F$ from β provided that x does not occur free in β. The following result shows that our evaluation semantics subsumes this method:

Corollary 6.36 *If $\beta \vdash_{\mathcal{CND}} F$ and $x \notin FV(\beta)$ then $\beta \vdash_{\mathcal{CND}} (\forall\, x)\, F$.*

Proof: By supposition, there is a D such that $\beta \vdash D \leadsto F$. Let

$$D' = \textbf{pick-any } x \textbf{ in } D$$

and let y be any variable that does not occur in $\beta \cup \{F\}$ or in D. By Corollary 6.35,

$$\beta \vdash \{x \mapsto y\}\, D \leadsto \{x \mapsto y\}\, F.$$

Hence, by $[R_8]$,

$$\beta \vdash D' \leadsto (\forall\, y)\, \{x \mapsto y\}\, F. \tag{6.53}$$

By Lemma 6.1,

$$(\forall\, x)\, F \approx_\alpha (\forall\, y)\, \{x \mapsto y\}\, F$$

so the desired conclusion follows from 6.53. ■

Theorem 6.37 (Dilution) *If $\beta \vdash D \leadsto F$ then $\beta \cup \beta' \vdash D \leadsto F$.*

Proof: By structural induction on D. When D is a claim or a primitive deduction the result is immediate by the semantics of claims, the primitive rules, and **specialize** and **ex-generalize**. When it is a hypothetical deduction **assume** H **in** D', we have $F = H \Rightarrow G$ and $\beta \cup \{H\} \vdash D' \leadsto G$, for some G. Inductively, $\beta \cup \beta' \cup \{H\} \vdash D' \leadsto G$, thus

$$\beta \cup \beta' \vdash \textbf{assume } H \textbf{ in } D' \leadsto H \Rightarrow G = F.$$

When D is a composition $D_1; D_2$ we have $\beta \vdash D_1 \leadsto G$ and $\beta \cup \{G\} \vdash D_2 \leadsto F$, for some G. Inductively, $\beta \cup \beta' \vdash D_1 \leadsto G$ and $\beta \cup \beta' \cup \{G\} \vdash D_2 \leadsto F$, thus

$$\beta \cup \beta' \vdash D_1; D_2 \leadsto F.$$

Next, when D is of the form **pick-any** x **in** D', we must have

$$\beta \vdash \{x \mapsto z\}\, D' \leadsto G \tag{6.54}$$

for some G, and

$$F = (\forall\, z)\, G \tag{6.55}$$

for a z that does not occur in β or in D'. Let w be a variable other than z that does not occur in

$$\beta \cup \beta' \cup \{F\}$$

or in D'. By Corollary 6.34 and 6.54 we get

$$\beta \vdash \{x \mapsto w\}\, D' \rightsquigarrow \{z \mapsto w\}\, G \tag{6.56}$$

and, from the inductive hypothesis,

$$\beta \cup \beta' \vdash \{x \mapsto w\}\, D' \rightsquigarrow \{z \mapsto w\}\, G. \tag{6.57}$$

Owing to our choice of w, $[R_8]$ and 6.57 give

$$\beta \cup \beta' \vdash \textbf{pick-any } x \textbf{ in } D' \rightsquigarrow (\forall\, w)\, \{z \mapsto w\}\, G. \tag{6.58}$$

But w does not occur in G, hence $F = (\forall\, z)\, G$ and $(\forall\, w)\, \{z \mapsto w\}\, G$ are identical (alphabetically equivalent—Lemma 6.1), and 6.58 becomes $\beta \cup \beta' \vdash D \rightsquigarrow F$.

Finally, suppose that D is of the form **pick-witness** x **for** $(\exists\, y)\, H$ **in** D'. Then we must have

$$(\exists\, y)\, H \in \beta \tag{6.59}$$

and

$$\beta \cup \{\{y \mapsto z\}\, H\} \vdash \{x \mapsto z\}\, D' \rightsquigarrow F \tag{6.60}$$

for some $z \notin FV(F)$ that does not occur in β or in D'. Let w be a variable other than z that does not occur in

$$\beta \cup \beta' \cup \{F\}$$

or in D'. By 6.60 and Theorem 6.33 we get

$$\{z \mapsto w\}\, \beta \cup \{\{z \mapsto w\}\, \{y \mapsto z\}\, H\} \vdash \{z \mapsto w\}\, \{x \mapsto z\}\, D' \rightsquigarrow \{z \mapsto w\}\, F. \tag{6.61}$$

Now $\{z \mapsto w\}\, \beta = \beta$ (since z does not occur in β), while

$$\{z \mapsto w\}\, \{y \mapsto z\}\, H = \{y \mapsto w\}\, H$$

(by Lemma 6.17, since z does not occur in H),

$$\{z \mapsto w\}\, \{x \mapsto z\}\, D' = \{x \mapsto w\}\, D'$$

(by Lemma 6.19, since z does not occur in D'), and

$$\{z \mapsto w\}\, F = F$$

(since $z \notin FV(F)$). Thus 6.61 becomes

$$\beta \cup \{\{y \mapsto w\}\, H\} \vdash \{x \mapsto w\}\, D' \rightsquigarrow F \tag{6.62}$$

$$\frac{}{D \equiv_\epsilon D}\;[E_1] \qquad \frac{D_1 \equiv_\epsilon D_3 \qquad D_2 \equiv_\epsilon D_4}{D_1; D_2 \equiv_\epsilon D_3; D_4}\;[E_2]$$

$$\frac{D_1 \equiv_\epsilon D_2}{\textbf{assume } F \textbf{ in } D_1 \equiv_\epsilon \textbf{assume } F \textbf{ in } D_2}\;[E_3]$$

$$\frac{\{x_1 \mapsto y\}\, D_1 \equiv_\epsilon \{x_2 \mapsto y\}\, D_2}{\textbf{pick-any } x_1 \textbf{ in } D_1 \equiv_\epsilon \textbf{pick-any } x_2 \textbf{ in } D_2}\;[E_4]$$

where y does not occur in D_1, D_2, $y \neq \{x_1, x_2\}$.

$$\frac{\{x_1 \mapsto y\}\, D_1 \equiv_\epsilon \{x_2 \mapsto y\}\, D_2}{\textbf{pick-witness } x_1 \textbf{ for } F \textbf{ in } D_1 \equiv_\epsilon \textbf{pick-witness } x_2 \textbf{ for } F \textbf{ in } D_2}\;[E_5]$$

where y does not occur in D_1, D_2, F, $y \neq \{x_1, x_2\}$.

Figure 6.5: Definition of the eigenvariance relation $\equiv_\epsilon$.

and the inductive hypothesis gives

$$\beta \cup \beta' \cup \{\{y \mapsto w\}\, H\} \vdash \{x \mapsto w\}\, D' \leadsto F. \tag{6.63}$$

Owing to our choice of w, 6.63 and $[R_9]$ give

$$\beta \cup \beta' \vdash \textbf{pick-witness } x \textbf{ for } (\exists\, y)\, H \textbf{ in } D' \leadsto F$$

which is the desired $\beta \cup \beta' \vdash D \leadsto F$. ∎

Next we introduce the relation of *eigenvariance*, denoted by $\equiv_\epsilon$. Intuitively, $D_1 \equiv_\epsilon D_2$ (D_1 and D_2 are "eigenvariant") if they only differ in the names of their eigenvariables. The precise definition appears in Figure 6.5. Eigenvariance is similar to the relation of alphabetic equivalence for formulas, or α-convertibility in the λ-calculus. Like those other relations, it is an equivalence, and it fully preserves the relevant semantics, as Theorem 6.45 below will show. We will also see that we can systematically "rename" the eigenvariables of any deduction to our liking.

Lemma 6.40 below is an important technical tool. We first need to prove analogous forms of the lemma for terms and formulas.

Lemma 6.38 *If* $\overline{\{x_1 \mapsto z\}}(s) = \overline{\{x_2 \mapsto z\}}(t)$ *for some* $z \notin Var(s) \cup Var(t)$, *then* $\overline{\{x_1 \mapsto z'\}}(s) = \overline{\{x_2 \mapsto z'\}}(t)$ *for all* z'.

Proof: By induction on s. When s is a variable there are two possible cases:

(a) $s = x_1$: Then $\{x_1 \mapsto z\}\, s = z = \{x_2 \mapsto z\}\, t$, and thus we must have $t = x_2$, for otherwise the equation $\{x_2 \mapsto z\}\, t = z$ would entail $t = z$, which is impossible by the supposition $z \notin \mathit{Var}(t)$. But if $t = x_2$ and $s = x_1$ then we clearly have $\{x_1 \mapsto z'\}\, s = \{x_2 \mapsto z'\}\, t$.

(b) $s \neq x_1$: Then
$$\{x_1 \mapsto z\}\, s = s = \{x_2 \mapsto z\}\, t \tag{6.64}$$
and thus we must have $t \neq x_2$, for if $t = x_2$ then $\{x_2 \mapsto z\}\, t = z$ and, by 6.64, $z = s$, which is impossible by the supposition $z \notin \mathit{Var}(s)$. But $t \neq x_2$ entails $\{x_2 \mapsto z\}\, t = t$, so, by 6.64, $t = s$. Accordingly,
$$\begin{aligned}\{x_1 \mapsto z'\}\, s &= t \\ &= t \qquad\qquad (\text{since } t \neq x_2) \\ &= \{x_2 \mapsto z'\}\, t.\end{aligned}$$

Thus the identity $\{x_1 \mapsto z'\}\, s = \{x_2 \mapsto z'\}\, t$ holds in both cases.

When s is a constant we must have $s = t$, so the result is immediate. Finally, when s is of the form $f(s_1, \ldots, s_n)$, t must be of the form $f(t_1, \ldots, t_n)$, so by supposition,
$$\{x_1 \mapsto z\}\, s_i = \{x_2 \mapsto z\}\, t_i \tag{6.65}$$
for $i = 1, \ldots, n$. Now
$$\{x_1 \mapsto z'\}\, s = f(\{x_1 \mapsto z'\}\, s_1, \ldots, \{x_1 \mapsto z'\}\, s_n)$$
and
$$\{x_2 \mapsto z'\}\, t = f(\{x_2 \mapsto z'\}\, t_1, \ldots, \{x_2 \mapsto z'\}\, t_n)$$
so now the result follows from (6.65) and the inductive hypothesis, since
$$z \notin \mathit{Var}(s_i) \cup \mathit{Var}(t_i)$$
for every $i \in \{1, \ldots, n\}$. ■

Using Lemma 6.38 for the base case, the following can be proved with a straightforward induction on F:

Lemma 6.39 *If $\{x_1 \mapsto z\}\, F = \{x_2 \mapsto z\}\, G$ for some z that does not occur in F or in G, then $\{x_1 \mapsto z'\}\, F = \{x_2 \mapsto z'\}\, G$ for every z' that does not occur in F or in G.*

Finally, we generalize to deductions as follows:

Lemma 6.40 *If* $\{x_1 \mapsto z\}\, D_1 \equiv_\epsilon \{x_2 \mapsto z\}\, D_2$ *for some* z *that does not occur in* D_1 *or in* D_2*, then* $\{x_1 \mapsto z'\}\, D_1 \equiv_\epsilon \{x_2 \mapsto z'\}\, D_2$ *for every* z' *that does not occur in* D_1 *or in* D_2*.*

Proof: We will use induction on the structure of D_1 to show that if

$$\theta_1\, D_1 \equiv_\epsilon \theta_2\, D_2 \tag{6.66}$$

then

$$\theta_1'\, D_1 \equiv_\epsilon \theta_2'\, D_2$$

where $\theta_1 = \{x_1 \mapsto z\}$, $\theta_2 = \{x_2 \mapsto z\}$, $\theta_1' = \{x_1 \mapsto z'\}$, and $\theta_2' = \{x_2 \mapsto z'\}$.

When D_1 is a claim F, D_2 must also be a claim G, and 6.66 entails

$$\theta_1\, F = \theta_2\, G.$$

Thus Lemma 6.39 yields $\theta_1'\, F = \theta_2'\, G$, i.e., $\theta_1'\, D_1 \equiv_\epsilon \theta_2'\, D_2$.

When D_1 is a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$, D_2 must also be of the form *Prim-Rule* $G_1, \ldots, G_n$, and 6.66 entails

$$\theta_1\, F_i = \theta_2\, G_i$$

for $i = 1, \ldots, n$. Thus Lemma 6.39 gives $\theta_1'\, F_i = \theta_2'\, G_i$, and hence $\theta_1'\, D_1 = \theta_2'\, D_2$. Consequently, $\theta_1'\, D_1 \equiv_\epsilon \theta_2'\, D_2$.

When D_1 is of the form **specialize** F **with** s, D_2 must be of the form

specialize G **with** t

and 6.66 implies $\theta_1\, F = \theta_2\, G$ and

$$\overline{\theta_1}(s) = \overline{\theta_2}(t).$$

Accordingly, Lemma 6.38 and Lemma 6.39 yield $\overline{\theta_1'}(s) = \overline{\theta_2'}(t)$ and $\theta_1'\, F = \theta_2'\, G$, so that $\theta_1'\, D_1 = \theta_2'\, D_2$ and $\theta_1'\, D_1 \equiv_\epsilon \theta_2'\, D_2$. The reasoning is similar when D_1 is of the form **ex-generalize** F **from** s.

If D_1 is of the form **assume** F **in** D_3, D_2 must be of the form **assume** G **in** D_4, and 6.66 entails

$$\begin{aligned} \theta_1\, F &= \theta_2\, G &(6.67)\\ \theta_1\, D_3 &\equiv_\epsilon \theta_2\, D_4. &(6.68) \end{aligned}$$

Inductively, 6.68 gives $\theta_1'\, D_3 \equiv_\epsilon \theta_2'\, D_4$, while 6.67 and Lemma 6.39 give $\theta_1'\, F = \theta_2'\, G$, hence, by $[E_3]$,

$$\textbf{assume } \theta_1'\, F \textbf{ in } \theta_1'\, D_3 \equiv_\epsilon \textbf{assume } \theta_2'\, G \textbf{ in } \theta_2'\, D_4$$

i.e., $\theta'_1\, D_1 \equiv_\epsilon \theta'_2\, D_2$.

When D_1 is of the form $D^1_l; D^1_r$, D_2 must be of the form $D^2_l; D^2_r$, and 6.66 entails

$$\theta_1\, D^1_l \quad \equiv_\epsilon \quad \theta_2\, D^2_l \tag{6.69}$$

$$\theta_1\, D^1_r \quad \equiv_\epsilon \quad \theta_2\, D^2_r. \tag{6.70}$$

Inductively, 6.69 and 6.70 give $\theta'_1\, D^1_l \equiv_\epsilon \theta'_2\, D^2_l$ and $\theta'_1\, D^1_r \equiv_\epsilon \theta'_2\, D^2_r$, hence, by $[E_2]$,

$$\theta'_1\, D^1_l; \theta'_1\, D^1_r \equiv_\epsilon \theta'_2\, D^2_l; \theta'_2\, D^2_r$$

which is to say, $\theta'_1\, D_1 = \theta'_2\, D_2$.

Next, suppose that D_1 is of the form **pick-any** y_1 **in** D'_1, so that D_2 must be of the form **pick-any** y_2 **in** D'_2. By supposition, we have

$$z \not\in \{y_1, y_2\} \tag{6.71}$$

and

$$z' \not\in \{y_1, y_2\}. \tag{6.72}$$

Further, the supposition $\theta_1\, D_1 \equiv_\epsilon \theta_2\, D_2$ means that

$$\{y_1 \mapsto w\}\, \theta_1[y_1 \mapsto y_1]\, D'_1 \equiv_\epsilon \{y_2 \mapsto w\}\, \theta_2[y_2 \mapsto y_2]\, D'_2 \tag{6.73}$$

for some $w \not\in \{y_1, y_2\}$ that does not occur in $\theta_1[y_1 \mapsto y_1]\, D'_1$ or in $\theta_2[y_2 \mapsto y_2]\, D'_2$. Now let v be a variable that does not occur in D_1 or in D_2, and such that

$$v \not\in \{x_1, x_2, z, z', w\}. \tag{6.74}$$

From 6.73 and the inductive hypothesis we get

$$\{y_1 \mapsto v\}\, \theta_1[y_1 \mapsto y_1]\, D'_1 \equiv_\epsilon \{y_2 \mapsto v\}\, \theta_2[y_2 \mapsto y_2]\, D'_2 \tag{6.75}$$

i.e.,

$$\{y_1 \mapsto v\}\, \{x_1 \mapsto z\}[y_1 \mapsto y_1]\, D'_1 \equiv_\epsilon \{y_2 \mapsto v\}\, \{x_2 \mapsto z\}[y_2 \mapsto y_2]\, D'_2. \tag{6.76}$$

We now observe that

$$\{y_1 \mapsto v\}\, \{x_1 \mapsto z\}[y_1 \mapsto y_1]\, D'_1 = \{x_1 \mapsto z\}[y_1 \mapsto y_1]\, \{y_1 \mapsto v\}\, D'_1 \tag{6.77}$$

as there are two cases: either $y_1 = x_1$ or not. If $y_1 = x_1$ then $\{x_1 \mapsto z\}[y_1 \mapsto y_1]$ is the empty substitution $\{\}$, and 6.77 holds trivially. Otherwise, if $y_1 \neq x_1$, then $\{x_1 \mapsto z\}[y_1 \mapsto y_1] = \{x_1 \mapsto z\}$, and 6.77 becomes

$$\{y_1 \mapsto v\}\, \{x_1 \mapsto z\}\, D'_1 = \{x_1 \mapsto z\}\, \{y_1 \mapsto v\}\, D'_1$$

which follows from Lemma 6.14 since (by 6.74, 6.71, and the inequality $y_1 \neq x_1$) the substitutions $\{y_1 \mapsto v\}$ and $\{x_1 \mapsto z\}$ are disjoint.

By parity of reasoning,

$$\{y_2 \mapsto v\}\,\{x_2 \mapsto z\}[y_2 \mapsto y_2]\,D_2' = \{x_2 \mapsto z\}[y_2 \mapsto y_2]\,\{y_2 \mapsto v\}\,D_2'. \tag{6.78}$$

Applying 6.77 and 6.78 to 6.76 yields

$$\{x_1 \mapsto z\}[y_1 \mapsto y_1]\,\{y_1 \mapsto v\}\,D_1' \equiv_\epsilon \{x_2 \mapsto z\}[y_2 \mapsto y_2]\,\{y_2 \mapsto v\}\,D_2'. \tag{6.79}$$

By applying Corollary 6.29 to both sides of 6.79 we get:

$$\{x_1 \mapsto z\}\,\{y_1 \mapsto v\}\,D_1' \equiv_\epsilon \{x_2 \mapsto z\}\,\{y_2 \mapsto v\}\,D_2'$$

and thus the inductive hypothesis yields

$$\{x_1 \mapsto z'\}\,\{y_1 \mapsto v\}\,D_1' \equiv_\epsilon \{x_2 \mapsto z'\}\,\{y_2 \mapsto v\}\,D_2'. \tag{6.80}$$

Now applying Corollary 6.29 to 6.80 gives

$$\{x_1 \mapsto z'\}[y_1 \mapsto y_1]\,\{y_1 \mapsto v\}\,D_1' \equiv_\epsilon \{x_2 \mapsto z'\}[y_2 \mapsto y_2]\,\{y_2 \mapsto v\}\,D_2' \tag{6.81}$$

and by establishing 6.77 and 6.78 with z' in place of z (with the same reasoning, but using 6.72 instead of 6.71), 6.81 becomes

$$\{y_1 \mapsto v\}\,\{x_1 \mapsto z'\}[y_1 \mapsto y_1]\,D_1' \equiv_\epsilon \{y_2 \mapsto v\}\,\{x_2 \mapsto z'\}[y_2 \mapsto y_2]\,D_2'. \tag{6.82}$$

By our choice of v and rule $[E_4]$, 6.82 entails

$$\textbf{pick-any } y_1 \textbf{ in } \{x_1 \mapsto z'\}[y_1 \mapsto y_1]\,D_1' \equiv_\epsilon \textbf{pick-any } y_2 \textbf{ in } \{x_2 \mapsto z'\}[y_2 \mapsto y_2]\,D_2'$$

which is to say $\{x_1 \mapsto z'\}\,D_1 \equiv_\epsilon \{x_2 \mapsto z'\}\,D_2$.

The reasoning is the same when D is of the form **pick-witness** y_1 **for** F_1 **in** D_1', and the induction is thus complete. ■

Next we show that eigenvariance is preserved by safe substitutions:

Lemma 6.41 *If $D_1 \equiv_\epsilon D_2$ and θ is safe for D_1 and D_2, then $\theta\,D_1 \equiv_\epsilon \theta\,D_2$.*

Proof: We use induction on the structure of D_1. When D_1 is a claim or a primitive deduction we must have $D_1 = D_2$, so $\theta\,D_1 \equiv_\epsilon \theta\,D_2$ is immediate. When D_1 is of the form **assume** F **in** D_1', D_2 must be of the form **assume** F **in** D_2', where $D_1' \equiv_\epsilon D_2'$. Thus, inductively, $\theta\,D_1' \equiv_\epsilon \theta\,D_2'$, and hence

$$\textbf{assume } \theta\,F \textbf{ in } \theta\,D_1' \equiv_\epsilon \textbf{assume } \theta\,F \textbf{ in } \theta\,D_2'$$

i.e., $\theta\, D_1 \equiv_\epsilon \theta\, D_2$.

When D_1 is a composition $D^1_l; D^1_r$, D_2 must be of the form $D^2_l; D^2_r$, and we must have

$$D^1_l \equiv_\epsilon D^2_l \tag{6.83}$$

and

$$D^1_r \equiv_\epsilon D^2_r. \tag{6.84}$$

From the inductive hypothesis, 6.83 and 6.84 yield

$$\theta\, D^1_l \equiv_\epsilon \theta\, D^2_l \tag{6.85}$$

and

$$\theta\, D^1_r \equiv_\epsilon \theta\, D^2_r \tag{6.86}$$

hence, from $[E_2]$,

$$\theta\, D^1_l; \theta\, D^1_r \equiv_\epsilon \theta\, D^2_l; \theta\, D^2_r$$

i.e., $\theta\, D_1 \equiv_\epsilon \theta\, D_2$.

Next, suppose that D_1 is of the form **pick-any** x_1 **in** D'_1, so that D_2 must be of the form **pick-any** x_2 **in** D'_2. The supposition $D_1 \equiv_\epsilon D_2$ then entails

$$\{x_1 \mapsto z\}\, D'_1 \equiv_\epsilon \{x_2 \mapsto z\}\, D'_2 \tag{6.87}$$

for some z that does not occur in D_1 or in D_2. Now let w be a variable that does not occur in D_1, D_2, or in $\mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta)$. By Lemma 6.38 and 6.87 we get

$$\{x_1 \mapsto w\}\, D'_1 \equiv_\epsilon \{x_2 \mapsto w\}\, D'_2. \tag{6.88}$$

Inductively, 6.88 gives

$$\theta\,\{x_1 \mapsto w\}\, D'_1 \equiv_\epsilon \theta\,\{x_2 \mapsto w\}\, D'_2 \tag{6.89}$$

and by Corollary 6.29 we get

$$\theta[x_1 \mapsto x_1]\,\{x_1 \mapsto w\}\, D'_1 \equiv_\epsilon \theta[x_2 \mapsto x_2]\,\{x_2 \mapsto w\}\, D'_2. \tag{6.90}$$

By our choice of w and the safety of θ for D_1, the substitutions $\theta[x_1 \mapsto x_1]$ and $\{x_1 \mapsto w\}$ are disjoint, and hence, by Lemma 6.14,

$$\theta[x_1 \mapsto x_1]\,\{x_1 \mapsto w\}\, D'_1 = \{x_1 \mapsto w\}\,\theta[x_1 \mapsto x_1]\, D'_1. \tag{6.91}$$

Likewise, $\theta[x_2 \mapsto x_2]$ and $\{x_2 \mapsto w\}$ are disjoint, therefore

$$\theta[x_2 \mapsto x_2]\,\{x_2 \mapsto w\}\, D'_2 = \{x_2 \mapsto w\}\,\theta[x_2 \mapsto x_2]\, D'_2. \tag{6.92}$$

Applying the identities 6.91 and 6.92 to 6.90 yields

$$\{x_1 \mapsto w\}\,\theta[x_1 \mapsto x_1]\,D_1' \equiv_\epsilon \{x_2 \mapsto w\}\,\theta[x_2 \mapsto x_2]\,D_2' \tag{6.93}$$

and by our choice of w and $[E_3]$ we get

$$\textbf{pick-any}\ x_1\ \textbf{in}\ \theta[x_1 \mapsto x_1]\,D_1' \equiv_\epsilon \textbf{pick-any}\ x_2\ \textbf{in}\ \theta[x_2 \mapsto x_2]\,D_2'$$

which is to say $\theta\,D_1 \equiv_\epsilon \theta\,D_2$. The case of existential instantiations is similar, and the induction is thus complete. ■

Lemma 6.42 *If $D_1 \equiv_\epsilon D$ and $D_2 \equiv_\epsilon D$ then $D_1 \equiv_\epsilon D_2$.*

Proof: We will use induction on the structure of D. When D is a non-compound deduction, $D_1 \equiv_\epsilon D$ and $D_2 \equiv_\epsilon D$ imply $D_1 = D$ and $D_2 = D$, thus $D_1 \equiv_\epsilon D_2$ is immediate. When D is of the form **assume** F **in** D', the suppositions $D_1 \equiv_\epsilon D$ and $D_2 \equiv_\epsilon D$ entail that D_1 and D_2 are of the forms **assume** F **in** D_1' and **assume** F **in** D_2', respectively, where $D_1' \equiv_\epsilon D'$ and $D_2' \equiv_\epsilon D'$. Therefore, inductively, $D_1' \equiv_\epsilon D_2'$, and thus $D_1 \equiv_\epsilon D_2$.

When D is of the form $D_l; D_r$, the suppositions $D_1 \equiv_\epsilon D$ and $D_2 \equiv_\epsilon D$ entail that D_1 must be of the form $D_l^1; D_r^1$ and that D_2 must be of the form $D_l^2; D_r^2$, where

$$D_l^1 \equiv_\epsilon D_l, \tag{6.94}$$

$$D_l^2 \equiv_\epsilon D_l, \tag{6.95}$$

$$D_r^1 \equiv_\epsilon D_r, \tag{6.96}$$

and

$$D_r^2 \equiv_\epsilon D_r. \tag{6.97}$$

From 6.94, 6.95, and the inductive hypothesis we get $D_l^1 \equiv_\epsilon D_l^2$; while from 6.96, 6.97, and the inductive hypothesis we infer $D_r^1 \equiv_\epsilon D_r^2$. Therefore, $D_l^1; D_r^1 \equiv_\epsilon D_l^2; D_r^2$, which is to say $D_1 \equiv_\epsilon D_2$.

When D is of the form **pick-any** x **in** D', the supposition $D_1 \equiv_\epsilon D$ entails that D_1 must be of the form **pick-any** x_1 **in** D_1', where

$$\{x_1 \mapsto z_1\}\,D_1' \equiv_\epsilon \{x \mapsto z_1\}\,D' \tag{6.98}$$

for some z_1 that does not occur in D_1 or in D. Likewise, the supposition $D_2 \equiv_\epsilon D$ entails that D_2 must be of the form **pick-any** x_2 **in** D_2', where

$$\{x_2 \mapsto z_2\}\,D_2' \equiv_\epsilon \{x \mapsto z_2\}\,D' \tag{6.99}$$

for some z_2 that does not occur in D_2 or in D. Let w be a variable that does not occur in D_1, D_2, or in D. Then 6.98 and 6.99 in tandem with Lemma 6.38 give

$$\{x_1 \mapsto w\}\, D_1' \equiv_\epsilon \{x \mapsto w\}\, D'$$

and

$$\{x_2 \mapsto w\}\, D_2' \equiv_\epsilon \{x \mapsto w\}\, D'.$$

Hence, inductively,

$$\{x_1 \mapsto w\}\, D_1' \equiv_\epsilon \{x_2 \mapsto w\}\, D_2'$$

and by our choice of w and $[E_4]$ we get $D_1 \equiv_\epsilon D_2$. The reasoning is similar when D is of the form **pick-witness** x **for** F **in** D', and thus the induction is complete. ■

Corollary 6.43 *$\equiv_\epsilon$ is an equivalence relation.*

Proof: It is readily verified that any reflexive relation R such that $x\,R\,y$ whenever $x\,R\,z$ and $y\,R\,z$ is an equivalence. For $\equiv_\epsilon$, the latter property was shown by Lemma 6.42. Reflexivity is immediate by $[E_1]$. ■

The relation of *observational equivalence* for first-order $\mathcal{CND}$ deductions is defined just as before: for a given β, we define $D_1 \approx_\beta D_2$ to mean that

$$\beta \vdash D_1 \rightsquigarrow F \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow F$$

for all F. If $D_1 \approx_\beta D_2$ we say that D_1 and D_2 are observationally equivalent *in* β. General observational equivalence is again defined by quantifying over all assumption bases: $D_1 \approx D_2$ iff $D_1 \approx_\beta D_2$ for all β. The following result is the first-order analogue of Lemma 4.18:

Lemma 6.44 *If $D_1 \approx D_2$ then* **assume** F **in** $D_1 \approx$ **assume** F **in** D_2. *Furthermore,*

$$D_1; D_2 \approx D_3; D_4$$

whenever $D_1 \approx D_3$ and $D_2 \approx D_4$.

The next result confirms that eigenvariant deductions have the same meaning:

Theorem 6.45 *$\equiv_\epsilon \subset \approx$; i.e., eigenvariant deductions are observationally equivalent.*

Proof: We will use induction on the structure of D_1 to prove that if $D_1 \equiv_\epsilon D_2$ then $D_1 \approx D_2$. When D_1 is a claim or a primitive deduction, we must have $D_2 = D_1$ and observational equivalence is trivial. When D_1 is a hypothetical deduction **assume** F **in** D'_1, D_2 must be of the form **assume** F **in** D'_2, where $D'_1 \equiv_\epsilon D'_2$. Inductively, $D'_1 \approx D'_2$, and thus Lemma 6.44 gives

$$\textbf{assume } F \textbf{ in } D'_1 \approx \textbf{assume } F \textbf{ in } D'_2$$

i.e., $D_1 \approx D_2$. When D_1 is a composition $D_l; D_r$, the supposition $D_1 \equiv_\epsilon D_2$ means that D_2 is of the form $D'_l; D'_r$, and that $D_l \equiv_\epsilon D'_l$, $D_r \equiv_\epsilon D'_r$. Accordingly, the inductive hypothesis entails $D_l \approx D'_l$, $D_r \approx D'_r$, and thus Lemma 6.44 yields $D_l; D_r \approx D'_l; D'_r$, i.e., $D_1 \approx D_2$.

Next, suppose that D_1 is of the form **pick-any** x_1 **in** D'_1. Then $D_1 \equiv_\epsilon D_2$ means that D_2 must be of the form **pick-any** x_2 **in** D'_2, where

$$\{x_1 \mapsto z\}\, D'_1 \equiv_\epsilon \{x_2 \mapsto z\}\, D'_2 \tag{6.100}$$

for some z that does not occur in D_1 or in D_2. Now suppose that

$$\beta \vdash D_1 \rightsquigarrow F. \tag{6.101}$$

This means that

$$\beta \vdash \{x_1 \mapsto w\}\, D'_1 \rightsquigarrow G \tag{6.102}$$

and $F = (\forall\, w)\, G$ for some G and a w that does not occur in β or in D'_1. Choose a variable $v \notin \{z, w\}$ that does not occur in D_1, D_2, F, or in β. Then 6.102 and Corollary 6.34 give

$$\beta \vdash \{x_1 \mapsto v\}\, D'_1 \rightsquigarrow \{w \mapsto v\}\, G. \tag{6.103}$$

From 6.100 and Lemma 6.40 we have

$$\{x_1 \mapsto v\}\, D'_1 \equiv_\epsilon \{x_2 \mapsto v\}\, D'_2$$

hence from the inductive hypothesis and 6.103 we get

$$\beta \vdash \{x_2 \mapsto v\}\, D'_2 \rightsquigarrow \{w \mapsto v\}\, G.$$

Therefore, owing to our choice of v,

$$\beta \vdash \textbf{pick-any } x_2 \textbf{ in } D'_2 \rightsquigarrow (\forall\, v)\, \{w \mapsto v\}\, G. \tag{6.104}$$

But $(\forall\, v)\, \{w \mapsto v\}\, G$ and $F = (\forall\, w)\, G$ are identical (alphabetically equivalent, by Lemma 6.1), thus 6.104 becomes $\beta \vdash D_2 \rightsquigarrow F$. The converse implication follows by parity of reasoning, based on the fact that $\equiv_\epsilon$ is symmetric.

Finally, suppose that D_1 is of the form **pick-witness** x_1 **for** $(\exists\, y)\, H$ **in** D_1'. Then $D_1 \equiv_\epsilon D_2$ entails that D_2 is of the form **pick-witness** x_2 **for** $(\exists\, y)\, H$ **in** D_2', where

$$\{x_1 \mapsto z\}\, D_1' \equiv_\epsilon \{x_2 \mapsto z\}\, D_2' \tag{6.105}$$

for some z that does not occur in D_1 or in D_2. Now suppose that

$$\beta \vdash D_1 \leadsto F. \tag{6.106}$$

This means that $(\exists\, y)\, H \in \beta$ and

$$\beta \cup \{\{y \mapsto w\}\, H\} \vdash \{x_1 \mapsto w\}\, D_1' \leadsto F \tag{6.107}$$

for some w that does not occur in β or in D_1, and such that

$$w \notin FV(F). \tag{6.108}$$

Pick any $v \neq \{z, w\}$ that does not occur in β, D_1, D_2, or in F. Then 6.107 and Theorem 6.33 give

$$\{w \mapsto v\}\, \beta \cup \{\{w \mapsto v\}\, \{y \mapsto w\}\, H\} \vdash \{w \mapsto v\}\, \{x_1 \mapsto w\}\, D_1' \leadsto \{w \mapsto v\}\, F. \tag{6.109}$$

Since w does not occur in β, $\{w \mapsto v\}\, \beta = \beta$. Moreover, by virtue of the fact that w does not occur in H or in D_1', Lemma 6.17 and Lemma 6.19 imply

$$\begin{aligned} \{w \mapsto v\}\, \{y \mapsto w\}\, H &= \{y \mapsto v\}\, H \\ \{w \mapsto v\}\, \{x_1 \mapsto w\}\, D_1' &= \{x_1 \mapsto v\}\, D_1' \end{aligned}$$

and thus 6.109 becomes

$$\beta \cup \{\{y \mapsto v\}\, H\} \vdash \{x_1 \mapsto v\}\, D_1' \leadsto \{w \mapsto v\}\, F. \tag{6.110}$$

Now by 6.105 and our choice of v, Lemma 6.40 implies

$$\{x_1 \mapsto v\}\, D_1' \equiv_\epsilon \{x_2 \mapsto v\}\, D_2'$$

which, in tandem with 6.110 and the inductive hypothesis, yields

$$\beta \cup \{\{y \mapsto v\}\, H\} \vdash \{x_2 \mapsto v\}\, D_2' \leadsto \{w \mapsto v\}\, F. \tag{6.111}$$

But w does not occur free in F, hence $\{w \mapsto v\}\, F = F$, and 6.111 becomes

$$\beta \cup \{\{y \mapsto v\}\, H\} \vdash \{x_2 \mapsto v\}\, D_2' \leadsto F. \tag{6.112}$$

From our choice of v and the fact that $(\exists\, y)\, H \in \beta$, 6.112 and $[R_9]$ yield

$$\beta \vdash \textbf{pick-witness}\ x_2\ \textbf{for}\ (\exists\, y)\, H\ \textbf{in}\ D_2' \leadsto F$$

i.e., $\beta \vdash D_2 \leadsto F$. The converse direction is established with a symmetric argument, and the induction is thus complete. ■

Lemma 6.46 **pick-any** x **in** $D \equiv_\epsilon$ **pick-any** y **in** $\{x \mapsto y\}\, D$ *for any* y *that does not occur in* D.

Proof: Pick any z that does not occur in D or in $\{x, y\}$. By Lemma 6.19,

$$\{y \mapsto z\}\, \{x \mapsto y\}\, D = \{x \mapsto z\}\, D,$$

thus, by $[E_1]$, $\{x \mapsto z\}\, D \equiv_\epsilon \{y \mapsto z\}\, \{x \mapsto y\}\, D$, and by $[E_4]$,

$$\textbf{pick-any } x \textbf{ in } D \equiv_\epsilon \textbf{pick-any } y \textbf{ in } \{x \mapsto y\}\, D$$

owing to our choice of z. ■

Likewise:

Lemma 6.47 *If* y *does not occur in* D *then*

$$\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D \equiv_\epsilon \textbf{pick-witness } y \textbf{ for } F \textbf{ in } \{x \mapsto y\}\, D.$$

Proof: Choose a z that does not occur in D or in $\{x, y\}$. By Lemma 6.19,

$$\{y \mapsto z\}\, \{x \mapsto y\}\, D = \{x \mapsto z\}\, D,$$

thus $\{x \mapsto z\}\, D \equiv_\epsilon \{y \mapsto z\}\, \{x \mapsto y\}\, D$ and, by $[E_5]$,

$$\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D \equiv_\epsilon \textbf{pick-witness } y \textbf{ for } F \textbf{ in } \{x \mapsto y\}\, D$$

which again holds by our choice of z. ■

Lemma 6.48 *If* $D_1 \equiv_\epsilon D_2$ *then*

(a) **pick-any** x **in** $D_1 \equiv_\epsilon$ **pick-any** x **in** D_2*; and*

(b) **pick-witness** x **for** F **in** $D_1 \equiv_\epsilon$ **pick-witness** x **for** F **in** D_2.

Proof: Choose any variable y that does not occur in D_1 or in D_2. Then $\{x \mapsto y\}$ is safe for D_1 and D_2 and thus Lemma 6.41 in tandem with the hypothesis $D_1 \equiv_\epsilon D_2$ entails

$$\{x \mapsto y\}\, D_1 \equiv_\epsilon \{x \mapsto y\}\, D_2$$

which in turn implies (a) and (b) (by $[E_4]$ and $[E_5]$). ■

Lemma 6.49 (Eigenvariable renaming) *There is an algorithm that will take any deduction D and any finite set of variables X and will produce a deduction D' such that $D' \equiv_\epsilon D$ and $EV(D') \cap X = \emptyset$.*

We define such an algorithm as follows:

$\mathit{EigenRename}(D, X) = R(D)$
where

$R(\textbf{assume } F \textbf{ in } D) = \textbf{assume } F \textbf{ in } R(D)$

$R(D_1; D_2) = R(D_1); R(D_2)$

$R(\textbf{pick-any } x \textbf{ in } D) = \textbf{pick-any } y \textbf{ in } R(\{x \mapsto y\}\, D)$, for $y \notin \mathit{Var}(D) \cup X$

$R(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D) = \textbf{pick-witness } y \textbf{ for } F \textbf{ in } R(\{x \mapsto y\}\, D)$,
for $y \notin \mathit{Var}(D) \cup X$

$R(D) = D$

Note that the algorithm does not prescribe any specific way of choosing the variable y in the **pick-any** and **pick-witness** clauses; it only requires that y does not occur in D or in the given set X. This non-determinism can readily be removed once we impose some fixed computable well-ordering on the set of all variables; we can then always choose the smallest available variable with respect to this ordering. Finally, the following establishes Lemma 6.49:

Lemma 6.50 *Let $D' = \mathit{EigenRename}(D, X)$. Then $EV(D') \cap X = \emptyset$ and $D' \equiv_\epsilon D$.*

Proof: We prove both assertions for $D' = R(D)$, for any fixed value of X. We will use induction on D. The only interesting cases are when D is a universal generalization **pick-any** x **in** D_b or an existential instantiation **pick-witness** x **for** F **in** D_b. In the former case we have $D' = \textbf{pick-any } y \textbf{ in } D'_b$, where $D'_b = R(\{x \mapsto y\}\, D_b)$ and y does not occur in D_b or in X. Inductively, we have $EV(D'_b) \cap X = \emptyset$, and since $y \notin X$, $EV(D') \cap X = \emptyset$. Moreover, because y does not occur in D_b, Lemma 6.46 yields

$$D = \textbf{pick-any } x \textbf{ in } D_b \equiv_\epsilon \textbf{pick-any } y \textbf{ in } \{x \mapsto y\}\, D_b. \tag{6.113}$$

Inductively, we have

$$\{x \mapsto y\}\, D_b \equiv_\epsilon R(\{x \mapsto y\}\, D_b) = D'_b$$

and thus Lemma 6.48 gives

$$\textbf{pick-any } y \textbf{ in } \{x \mapsto y\}\, D_b \equiv_\epsilon \textbf{pick-any } y \textbf{ in } R(\{x \mapsto y\}\, D_b) = D' \tag{6.114}$$

Now $D \equiv_\epsilon D'$ follows from 6.113 and 6.114 by the transitivity of $\equiv_\epsilon$. A similar argument applies to existential instantiations, using Lemma 6.47 instead of Lemma 6.46. ■

We will say that a deduction of the form

$$\textbf{pick-witness } x \textbf{ for } (\exists\, y)\, F \textbf{ in } D \tag{6.115}$$

is *normal* if the variable x does not occur free in the formula $(\exists\, y)\, F$. Non-normal deductions—where x does occur free in $(\exists\, y)\, F$—are in bad style because, intuitively, they seem to be saying "Pick as a witness *this* particular object". What makes x "particular", of course, is that it occurs free in the assumption base (recall that the formula $(\exists\, y)\, F$ must be in the assumption base during the evaluation of 6.115).

More generally, we will say that a deduction D is normal iff every subdeduction of it of the form 6.115 is normal. Because eigenvariables can be consistently renamed without changing the meaning of a deduction, from here on we will assume that all deductions are normal. We lose no generality in making this assumption, as the renaming algorithm we presented above can always be used to ensure that a deduction is normal.

Next, we define $\mathcal{C}(D)$, the *conclusion* of a deduction D:

$$\begin{aligned}
\mathcal{C}(F) &= F && (6.116)\\
\mathcal{C}(\textbf{specialize } (\forall\, x)\, F \textbf{ with } t) &= \{x \mapsto t\}\, F && (6.117)\\
\mathcal{C}(\textbf{ex-generalize } (\exists\, x)\, F \textbf{ from } t) &= (\exists\, x)\, F && (6.118)\\
\mathcal{C}(\textbf{assume } F \textbf{ in } D) &= F \Rightarrow \mathcal{C}(D) && (6.119)\\
\mathcal{C}(D_1; D_2) &= \mathcal{C}(D_2) && (6.120)\\
\mathcal{C}(\textbf{pick-any } x \textbf{ in } D) &= (\forall\, x)\, \mathcal{C}(D) && (6.121)\\
\mathcal{C}(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D) &= \mathcal{C}(D) && (6.122)
\end{aligned}$$

where the clauses for primitive deductions other than universal specializations and existential generalizations are as in the propositional case (see Section B.1). A straightforward induction on D will show:

Lemma 6.51 *If x does not occur in D then $x \notin Var(\mathcal{C}(D))$.*

Lemma 6.52 $\mathcal{C}(\theta\, D) = \theta\, \mathcal{C}(D)$.

Proof: By structural induction on D. When D is a claim F we have

$$\mathcal{C}(\theta\, D) = \mathcal{C}(\theta\, F) = \theta\, F = \theta\, \mathcal{C}(F) = \theta\, \mathcal{C}(D).$$

When D is a primitive deduction of the form *Prim-Rule* $F_1, \ldots, F_n$ the result is readily verified by an inspection of each possible case. We demonstrate the case where D is of the form **both** F_1, F_2:

$$\begin{array}{rcl}\mathcal{C}(\theta\, D) & = & \mathcal{C}(\theta\ \textbf{both}\ F_1, F_2) \\ & = & \mathcal{C}(\textbf{both}\ \theta\, F_1, \theta\, F_2) \\ & = & \theta\, F_1 \wedge \theta\, F_2 \\ & = & \theta\,(F_1 \wedge F_2) \\ & = & \theta\, \mathcal{C}(\textbf{both}\ F_1, F_2) \\ & = & \theta\, \mathcal{C}(D).\end{array}$$

Next, suppose that D is a universal specialization of the form

$$\textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t.$$

By alphabetic conversion, we lose no generality in assuming

$$x \notin \mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta) \tag{6.123}$$

so that $\theta[x \mapsto x] = \theta$ and $\theta\,(\forall\, x)\, F = (\forall\, x)\, \theta\, F$. We then reason as follows:

$$\begin{array}{rcll}\mathcal{C}(\theta\, D) & = & \mathcal{C}(\textbf{specialize}\ (\forall\, x)\, \theta\, F\ \textbf{with}\ \overline{\theta}(t)) & \\ & = & \{x \mapsto \overline{\theta}(t)\}\, \theta\, F & \text{(6.123 and Corollary 6.32)} \\ & = & \theta\, \{x \mapsto t\}\, F & \\ & = & \theta\, \mathcal{C}(\textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t) & \\ & = & \theta\, \mathcal{C}(D). & \end{array}$$

When D is an existential generalization **ex-generalize** F **from** t we have:

$$\begin{array}{rcl}\mathcal{C}(\theta\, D) & = & \mathcal{C}(\textbf{ex-generalize}\ \theta\, F\ \textbf{from}\ \overline{\theta}(t)) \\ & = & \theta\, F \\ & = & \theta\, \mathcal{C}(\textbf{ex-generalize}\ F\ \textbf{from}\ t) \\ & = & \theta\, \mathcal{C}(D).\end{array}$$

When D is a hypothetical deduction **assume** F **in** D' we have:

$$\begin{array}{rcll}\mathcal{C}(\theta\, D) & = & \mathcal{C}(\textbf{assume}\ \theta\, F\ \textbf{in}\ \theta\, D') & \\ & = & \theta\, F \Rightarrow \mathcal{C}(\theta\, D') & \text{(from the inductive hypothesis)} \\ & = & \theta\, F \Rightarrow \theta\, \mathcal{C}(D') & \\ & = & \theta\,(F \Rightarrow \mathcal{C}(D')) & \\ & = & \theta\, \mathcal{C}(\textbf{assume}\ F\ \textbf{in}\ D') & \\ & = & \theta\, \mathcal{C}(D). & \end{array}$$

When D is a composition $D_1; D_2$:

$$\begin{array}{rcll} \mathcal{C}(\theta\, D) & = & \mathcal{C}(\theta\, D_1; \theta\, D_2) & \\ & = & \mathcal{C}(\theta\, D_2) & \text{(from the inductive hypothesis)} \\ & = & \theta\, \mathcal{C}(D_2) & \\ & = & \theta\, \mathcal{C}(D_1; D_2) & \\ & = & \theta\, \mathcal{C}(D). & \end{array}$$

When D is a universal generalization **pick-any** x **in** D':

$$\begin{array}{rcll} \mathcal{C}(\theta\, D) & = & \mathcal{C}(\textbf{pick-any}\ x\ \textbf{in}\ \theta[x \mapsto x]\, D') & \\ & = & (\forall\, x)\, \mathcal{C}(\theta[x \mapsto x]\, D') & \text{(from the inductive hypothesis)} \\ & = & (\forall\, x)\, \theta[x \mapsto x]\, \mathcal{C}(D') & \\ & = & \theta\, (\forall\, x)\, \mathcal{C}(D') & \\ & = & \theta\, \mathcal{C}(D). & \end{array}$$

Finally, suppose that D is an existential instantiation

$$\textbf{pick-witness}\ x\ \textbf{for}\ F\ \textbf{in}\ D'.$$

By Lemma 6.49, we may assume $x \notin \mathit{Supp}(\theta)$, so that

$$\theta[x \mapsto x] = \theta. \tag{6.124}$$

Thus:

$$\begin{array}{rcll} \mathcal{C}(\theta\, D) & = & \mathcal{C}(\textbf{pick-witness}\ x\ \textbf{for}\ \theta\, F\ \textbf{in}\ \theta[x \mapsto x]\, D') & \text{(by 6.124)} \\ & = & \mathcal{C}(\textbf{pick-witness}\ x\ \textbf{for}\ \theta\, F\ \textbf{in}\ \theta\, D') & \\ & = & \mathcal{C}(\theta\, D') & \text{(inductively)} \\ & = & \theta\, \mathcal{C}(D') & \\ & = & \theta\, \mathcal{C}(\textbf{pick-witness}\ x\ \textbf{for}\ F\ \textbf{in}\ D') & \\ & = & \theta\, \mathcal{C}(D). & \end{array}$$

The result now follows by structural induction. ∎

Theorem 6.53 *If $\beta \vdash D \leadsto F$ then $F = \mathcal{C}(D)$.*

Proof: By induction on the structure of D. When D is a claim the result is immediate. In the case of a primitive deduction *Prim-Rule* $F_1, \ldots, F_n$, the result is verified by a straightforward inspection of the different rules. When D is a universal specialization or an existential generalization, the result holds by virtue of $[R_6]$ and $[R_7]$. Hypothetical deductions and compositions are readily handled by applying the inductive hypothesis.

Next, suppose that D is of the form **pick-any** x **in** D'. Then the supposition $\beta \vdash D \leadsto F$ means that

$$\beta \vdash \{x \mapsto z\}\, D' \leadsto G \tag{6.125}$$

and

$$F = (\forall\, z)\, G \tag{6.126}$$

for some z that does not occur in β or in D'. Inductively, 6.125 gives

$$G = \mathcal{C}(\{x \mapsto z\}\, D'). \tag{6.127}$$

By Lemma 6.52,

$$G = \{x \mapsto z\}\, \mathcal{C}(D'). \tag{6.128}$$

Therefore, by 6.126,

$$F = (\forall\, z)\, \{x \mapsto z\}\, \mathcal{C}(D').$$

But, by Lemma 6.51, z does not occur in $\mathcal{C}(D')$, hence $(\forall\, x)\, \mathcal{C}(D')$ and

$$F = (\forall\, z)\, \{x \mapsto z\}\, \mathcal{C}(D')$$

are identical (α-convertible), and thus

$$F = (\forall\, x)\, \mathcal{C}(D') = \mathcal{C}(D).$$

Finally, suppose that D is an existential instantiation

pick-witness x **for** $(\exists\, y)\, H$ **in** D'.

Then the supposition $\beta \vdash D \leadsto F$ means that $(\exists\, y)\, H \in \beta$ and

$$\beta \cup \{\{y \mapsto z\}\, H\} \vdash \{x \mapsto z\}\, D' \leadsto F \tag{6.129}$$

for some z that does not occur in β or in D', and where $z \notin FV(F)$. Inductively, 6.129 gives $F = \mathcal{C}(\{x \mapsto z\}\, D')$. By Lemma 6.52,

$$\mathcal{C}(\{x \mapsto z\}\, D') = \{x \mapsto z\}\, \mathcal{C}(D')$$

hence

$$F = \{x \mapsto z\}\, \mathcal{C}(D')$$

and

$$\{z \mapsto x\}\, F = \{z \mapsto x\}\, \{x \mapsto z\}\, \mathcal{C}(D'). \tag{6.130}$$

But z does not occur in D', hence, by Lemma 6.51, z does not occur in $\mathcal{C}(D')$, and Corollary 6.18 yields

$$\{z \mapsto x\}\, \{x \mapsto z\}\, \mathcal{C}(D') = \mathcal{C}(D').$$

Thus 6.130 becomes

$$\{z \mapsto x\}\, F = \mathcal{C}(D'). \tag{6.131}$$

But z does not occur free in F, hence $\{z \mapsto x\}\, F = F$ and 6.131 gives

$$F = \mathcal{C}(D') = \mathcal{C}(D),$$

completing the induction. ■

Corollary 6.54 (Conclusion Uniqueness) *If $\beta_1 \vdash D \rightsquigarrow F_1$ and $\beta_2 \vdash D \rightsquigarrow F_2$ then $F_1 = F_2$.*

An interpreter that evaluates any (well-formed) deduction in any given assumption base appears in Figure 6.6. It is a proper extension of the propositional interpeter shown in Figure 4.4, augmented with clauses that handle universal and existential generalizations and instantiations. In order to obtain a deterministic algorithm we assume that there is a fixed well-ordering $\prec_{\mathcal{V}}$ on the set of all variables such that we can mechanically produce the $\prec_{\mathcal{V}}$-smallest element of any co-finite subset of $\mathcal{V}$ (e.g., this is trivial if we identify variables with natural numbers through some appropriate bijection). The same basic results carry over from the propositional setting: evaluation always terminates, respects the semantics (i.e., the interpreter is sound), and always produces a conclusion if one is derivable (i.e., the interpreter is complete).

The following is immediate by observing that the size of the input deduction strictly decreases with each recursive invocation of the interpreter:

Theorem 6.55 (Termination) *Eval always terminates. In particular, the recursion tree spawned by a call Eval(D, β) has size $\Theta(n)$, where n is the size of D.*

Theorem 6.56 *$\beta \vdash D \rightsquigarrow F$ iff Eval$(D, \beta) = F$. Therefore, by termination,*

$$\mathit{Eval}(D, \beta) = \mathit{error}$$

iff there is no F such that $\beta \vdash D \rightsquigarrow F$.

$$
\begin{array}{l}
\mathit{Eval}(D,\beta) = \mathit{ev}(D) \\
\text{where} \\
\quad \mathit{ev}(F) = F \in \beta \cup \{\textbf{true}, \neg\textbf{false}\} \to F, \mathit{error} \\
\quad \mathit{ev}(\mathit{Prim\text{-}Rule}\ F_1,\ldots,F_n) = \mathit{do\text{-}prim\text{-}rule}(\mathit{Prim\text{-}Rule}, [F_1,\ldots,F_n], \beta) \\
\quad \mathit{ev}(\textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t) = (\forall\, x)\, F \in \beta \to \{x \mapsto t\}\, F, \mathit{error} \\
\quad \mathit{ev}(\textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t) = \{x \mapsto t\}\, F \in \beta \to (\exists\, x)\, F, \mathit{error} \\
\quad \mathit{ev}(\textbf{assume}\ F\ \textbf{in}\ D) = \mathit{let}\ G = \mathit{Eval}(D, \beta \cup \{F\}) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ G = \mathit{error} \to \mathit{error}, F \Rightarrow G \\
\quad \mathit{ev}(D_1; D_2) = \mathit{let}\ F_1 = \mathit{Eval}(D_1, \beta) \\
\qquad\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad\quad\ \ F_1 = \mathit{error} \to \mathit{error}, \mathit{Eval}(D_2, \beta \cup \{F_1\}) \\
\quad \mathit{ev}(\textbf{pick-any}\ x\ \textbf{in}\ D) = \mathit{let}\ z = \text{the } \prec_{\mathcal{V}}\text{-least variable not in } \beta \text{ or in } D \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad G = \mathit{Eval}(\{x \mapsto z\}\, D, \beta) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ G = \mathit{error} \to \mathit{error}, (\forall\, z)\, G \\
\quad \mathit{ev}(\textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, H\ \textbf{in}\ D) = \\
\qquad (\exists\, y)\, H \notin \beta \to \mathit{error},\ \mathit{let}\ z = \text{the } \prec_{\mathcal{V}}\text{-least variable not in } \beta \text{ or in } D \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad F = \mathit{Eval}(\{x \mapsto z\}\, D, \beta \cup \{\{y \mapsto z\}\, H\}) \\
\qquad\qquad\qquad\qquad\qquad \mathit{in} \\
\qquad\qquad\qquad\qquad\qquad\ \ F = \mathit{error} \to \mathit{error}, (z \in FV(F) \to \mathit{error}, F)
\end{array}
$$

Figure 6.6: An interpreter for first-order $\mathcal{CND}$.

Proof: By induction on the structure of D. When D is a claim or a primitive deduction the result is verified by a straightforward inspection of the semantics of $\mathcal{CND}$. When D is a hypothetical deduction or a composition, the result follows readily from the inductive hypothesis. The only interesting cases are universal generalizations and existential instantiations.

Suppose first that D is of the form **pick-any** x **in** D' and that $\beta \vdash D \leadsto F$. This means that

$$\beta \vdash \{x \mapsto w\}\, D' \leadsto H \tag{6.132}$$

(where, by Theorem 6.53, $H = \mathcal{C}(\{x \mapsto w\}\, D'))$, and that

$$F = (\forall\, w)\, H \tag{6.133}$$

for some w that does not occur in D' or in β. Let z be the $\prec_{\mathcal{V}}$-least variable that does not occur in β or in D. There are two cases to consider:

(a) $z = w$: Then, from 6.132 and the inductive hypothesis we get

$$H = Eval(\{x \mapsto w\}\, D', \beta) = Eval(\{x \mapsto z\}\, D', \beta)$$

and, by the definition of *Eval* and 6.133,

$$Eval(D, \beta) = (\forall\, z)\, H = (\forall\, w)\, H = F.$$

(b) $z \neq w$: Then from 6.132 and Corollary 6.34,

$$\beta \vdash \{x \mapsto z\}\, D' \rightsquigarrow \{w \mapsto z\}\, H$$

and thus, inductively,

$$Eval(\{x \mapsto z\}\, D', \beta) = \{w \mapsto z\}\, H$$
$$Eval(D, \beta) = (\forall\, z)\, \{w \mapsto z\}\, H. \tag{6.134}$$

But

$$H = \mathcal{C}(\{x \mapsto w\}\, D')$$

and since z does not occur in D' and $z \neq w$, Lemma 6.23 implies that z does not occur in $\{x \mapsto w\}\, D'$, so that, by Lemma 6.51, z does not occur in

$$\mathcal{C}(\{x \mapsto w\}\, D') = H.$$

Therefore, by Lemma 6.1, $(\forall\, z)\, \{w \mapsto z\}\, H$ is identical (alphabetically equivalent) to $(\forall\, w)\, H = F$, so 6.134 becomes

$$Eval(D, \beta) = F.$$

The converse implication—that if $Eval(D, \beta) = F$ then $\beta \vdash D \rightsquigarrow F$—is readily established by the corresponding inductive hypothesis and rule $[R_8]$, owing to the choice of z as the $\prec_{\mathcal{V}}$-least variable not occuring in β or in D'.

Finally, suppose that D is of the form **pick-witness** x **for** $(\exists\, y)\, G$ **in** D' and that $\beta \vdash D \rightsquigarrow F$. Accordingly, we must have

$$(\exists\, y)\, G \in \beta \tag{6.135}$$

and

$$\beta \cup \{\{y \mapsto w\}\, G\} \vdash \{x \mapsto w\}\, D' \rightsquigarrow F \tag{6.136}$$

for some w that does not occur in β or in D', and where

$$w \notin FV(F). \tag{6.137}$$

Let z be the $\prec_{\mathcal{V}}$-least variable that does not occur in β or in D. Again we distinguish two cases:

(a) $z = w$: Then, by 6.136 and the inductive hypothesis,

$$Eval(\{x \mapsto z\}\, D', \beta \cup \{\{y \mapsto z\}\, G\}) = F$$

and thus from 6.137 and the definition of *Eval* we get $Eval(D, \beta) = F$.

(b) $z \neq w$: Then, by Lemma 6.23, z does not occur in $\{x \mapsto w\}\, D'$, and thus $\{w \mapsto z\}$ is safe for $\{x \mapsto w\}\, D'$. Accordingly, Theorem 6.33 and 6.136 yield

$$\{w \mapsto z\}\, \beta \cup \{\{w \mapsto z\}\, \{y \mapsto w\}\, G\} \vdash \{w \mapsto z\}\, \{x \mapsto w\}\, D' \rightsquigarrow \{w \mapsto z\}\, F. \quad (6.138)$$

Further, since w does not occur in β, $\{w \mapsto z\}\, \beta = \beta$; and since $w \notin FV(F)$, $\{w \mapsto z\}\, F = F$. Consequently, by Lemma 6.17 and Lemma 6.19, 6.138 becomes

$$\beta \cup \{\{y \mapsto z\}\, G\} \vdash \{x \mapsto z\}\, D' \rightsquigarrow F. \quad (6.139)$$

Inductively,

$$Eval(\{x \mapsto z\}\, D', \beta \cup \{\{y \mapsto z\}\, G\}) = F. \quad (6.140)$$

Since $F = \mathcal{C}(D')$ and z does not occur in D', Lemma 6.51 entails that z does not occur in F. Therefore, it follows from 6.140 and the definition of *Eval* that $Eval(D, \beta) = F$.

The converse implication—that $\beta \vdash D \rightsquigarrow F$ whenever $Eval(D, \beta) = F$—follows readily from the corresponding inductive hypothesis and $[R_9]$, by virtue of our choice of z. ■

Finally, we define the set of *free* (or *strict*) *assumptions* of a deduction D, denoted $FA(D)$, by conservatively extending the corresponding propositional definition in the following manner:

$FA(\textbf{true}) = FA(\neg\textbf{false}) = \emptyset$
$FA(F) = \{F\} \quad (F \notin \{\textbf{true}, \neg\textbf{false}\})$
$FA(\textbf{left-either}\ F_1, F_2) = \{F_1\}$
$FA(\textbf{right-either}\ F_1, F_2) = \{F_2\}$
$FA(\mathit{Prim\text{-}Rule}\ F_1, \ldots, F_n) = \{F_1, \ldots, F_n\} \quad (\mathit{Prim\text{-}Rule} \notin \{\textbf{left-either}, \textbf{right-either}\})$
$FA(\textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t) = \{(\forall\, x)\, F\}$
$FA(\textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t) = \{\{x \mapsto t\}\, F\}$
$FA(\textbf{assume}\ F\ \textbf{in}\ D) = FA(D) - \{F\}$
$FA(D_1; D_2) = FA(D_1) \cup [FA(D_2) - \{\mathcal{C}(D_1)\}]$
$FA(\textbf{pick-any}\ x\ \textbf{in}\ D) = \mathit{let}\ \ \Phi = FA(D)$
$\qquad\qquad\qquad\qquad\qquad \mathit{in}$
$\qquad\qquad\qquad\qquad\qquad\quad \Phi = \mathit{error} \rightarrow \mathit{error}, [x \in FV(\Phi) \rightarrow \mathit{error}, \Phi]$

$$
\begin{array}{l}
FA(\textbf{pick-witness } x \textbf{ for } (\exists\, y)\, F \textbf{ in } D) = \\
\quad x \in FV(\mathcal{C}(D)) \rightarrow error, \\
\qquad\qquad\qquad\quad let\ \ \Phi = FA(D) \\
\qquad\qquad\qquad\quad in \\
\qquad\qquad\qquad\qquad \Phi = error \rightarrow error, let\ \ \Psi = \Phi - \{\{y \mapsto x\}\, F\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad in \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ x \in FV(\Psi) \rightarrow error, \Psi \cup \{(\exists\, y)\, F\}
\end{array}
$$

The next key result is Theorem 6.65 below, asserting that $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ iff $\beta \supseteq FA(D)$. Note that when we write $\beta \supseteq FA(D)$ we tacitly imply that $FA(D) \neq error$. In general, we adopt the convention that in any context in which an expression such as $FA(D)$ would have to denote a set of formulas for some enclosing expression to be meaningful, we are tacitly conjoining the qualification $FA(D) \neq error$. Accordingly, the full content of the theorem in question is: $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ iff $FA(D) \neq error$ and $\beta \supseteq FA(D)$. Observe that this convention is not necessary for an identity such as $FA(D_1) = FA(D_2)$, as the values of $FA(D_1)$ and $FA(D_2)$ do not have to be sets for such an identity to be meaningful. In particular, this equality will be considered valid iff both $FA(D_1)$ and $FA(D_2)$ are *error*, or else both denote the same set of formulas.

We proceed with some auxiliary results that will be used in the proof of Theorem 6.65. The following is the analogue of Lemma 6.51:

Lemma 6.57 *If x does not occur in D then $x \notin Var(FA(D))$.*

Proof: Immediate by induction on D. ■

The next two lemmas are direct consequences our definitions. Note that the converse of part (b) of the following result does not hold, i.e., we do not in general have $\theta\,(\Phi_1 - \Phi_2) = \theta\,\Phi_1 - \theta\,\Phi_2$.

Lemma 6.58 (a) $\theta\,(\Phi_1 \cup \Phi_2) = \theta\,\Phi_1 \cup \theta\,\Phi_2$;

(b) $\theta\,\Phi_1 - \theta\,\Phi_2 \subseteq \theta\,(\Phi_1 - \Phi_2)$.

Lemma 6.59 *If $x \in FV(F)$ then $y \in FV(\{x \mapsto y\}\, F)$.*

Lemma 6.60 *If $x \notin Supp(\theta) \cup RanVar(\theta)$ then:*

(a) *$x \in FV(F)$ iff $x \in FV(\theta\, F)$;*

(b) *$x \in FV(\Phi)$ iff $x \in FV(\theta\, \Phi)$.*

Proof: Part (b) follows automatically from (a). One half of (a) follows directly from Lemma 6.22. The other half is a straightforward induction on the structure of F. ■

Lemma 6.61 *$x \notin FV(FA(D))$ whenever D is of the form* **pick-any** x **in** D' *or of the form* **pick-witness** x **for** F **in** D'.

Proof: By direct inspection of the definition of $FA(D)$. ■

Lemma 6.62 *If $\overline{\{x_1 \mapsto x_2\}}(s) = \overline{\{x_1 \mapsto x_2\}}(t)$ for $x_2 \notin Var(s) \cup Var(t)$ then $s = t$.*

Proof: By induction on s. When s is a variable z, there are two cases:

(1) $z = x_1$: Then

$$\{x_1 \mapsto x_2\}\, s = x_2 \tag{6.141}$$

so, by supposition, we must have

$$\{x_1 \mapsto x_2\}\, t = x_2. \tag{6.142}$$

Thus the assumption $x_2 \neq t$ entails

$$t = x_1 = z = s.$$

(2) $z \neq x_1$: Then

$$\{x_1 \mapsto x_2\}\, z = z. \tag{6.143}$$

We must also have $t \neq x_1$, for otherwise we would have

$$\{x_1 \mapsto x_2\}\, t = x_2 = \{x_1 \mapsto x_2\}\, z = z$$

which contradicts the assumption $x_2 \notin Var(s)$ (since $s = z$). But $t \neq x_1$ means that

$$\{x_1 \mapsto x_2\}\, t = t$$

and now the supposition $\{x_1 \mapsto x_2\}\, s = \{x_1 \mapsto x_2\}\, t$ in tandem with 6.143 implies $t = z = s$.

When s is a constant symbol the result is immediate. Finally, when s is an application $f(s_1, \ldots, s_n)$, t must be of the form $f(t_1, \ldots, t_n)$ and we must have

$$\{x_1 \mapsto x_2\}\, s_i = \{x_1 \mapsto x_2\}\, t_i$$

for $i = 1, \ldots, n$. Hence, inductively, $s_i = t_i$ and thus $s = t$. ■

Lemma 6.63 *If* $\{x_1 \mapsto x_2\}\, F = \{x_1 \mapsto x_2\}\, G$ *for* $x_2 \notin \mathit{Var}(F) \cup \mathit{Var}(G)$ *then* $F = G$.

Proof: By structural induction on F. When F is an atom $R(s_1, \ldots, s_n)$, the result follows from Lemma 6.62. When F is a formula built from a propositional constructor the result follows readily from the inductive hypothesis. For instance, when F is of the form $F_1 \circ F_2$, G must be of the form $G_1 \circ G_2$, and the assumption

$$\{x_1 \mapsto x_2\}\, F = \{x_1 \mapsto x_2\}\, G$$

entails

$$\{x_1 \mapsto x_2\}\, F_1 \circ \{x_1 \mapsto x_2\}\, F_2 = \{x_1 \mapsto x_2\}\, G_1 \circ \{x_1 \mapsto x_2\}\, G_2$$

so that

$$\{x_1 \mapsto x_2\}\, F_1 = \{x_1 \mapsto x_2\}\, G_1$$

and

$$\{x_1 \mapsto x_2\}\, F_2 = \{x_1 \mapsto x_2\}\, G_2.$$

By the inductive hypothesis, $F_1 = G_1$ and $F_2 = G_2$, therefore

$$F_1 \circ F_2 = G_1 \circ G_2$$

i.e., $F = G$.

Finally, when F is of the form $(Q\, x)\, F'$, G must also be of the form $(Q\, x)\, G'$, where we may assume by renaming that $x \neq x_1$. Now the supposition

$$\{x_1 \mapsto x_2\}\, F = \{x_1 \mapsto x_2\}\, G$$

entails

$$\{x_1 \mapsto x_2\}\, F' = \{x_1 \mapsto x_2\}\, G'$$

and the inductive hypothesis gives $F' = G'$. Therefore, $F = G$. ∎

Lemma 6.64 *If* w' *does not occur in* D *then* $\mathit{FA}(\{w \mapsto w'\}\, D) = \{w \mapsto w'\}\, \mathit{FA}(D)$.

Proof: We will prove

$$\mathit{FA}(\theta\, D) = \theta\, \mathit{FA}(D)$$

by induction on the structure of D, writing θ for $\{w \mapsto w'\}$. When D is the claim **true** or the claim $\neg$**false**, the result is immediate. When D is a claim

$$F \notin \{\mathbf{true}, \neg\mathbf{false}\}$$

we have

$$FA(\theta\, D) = FA(\theta\, F) = \{\theta\, F\} = \theta\,\{F\} = \theta\, FA(D).$$

When it is a primitive deduction

$$\textit{Prim-Rule}\ F_1, \ldots, F_n$$

the result is verified by a case analysis of *Prim-Rule*. We illustrate with **both**:

$$FA(\theta\, D) = FA(\textbf{both}\ \theta\, F_1, \theta\, F_2) = \{\theta\, F_1, \theta\, F_2\} = \theta\,\{F_1, F_2\} = \theta\, FA(D).$$

When D is of the form **specialize** $(\forall\, x)\, F$ **with** t we have:

$$\begin{aligned}
FA(\theta\, D) &= FA(\textbf{specialize}\ (\forall\, x)\, \theta[x \mapsto x]\, F\ \textbf{with}\ \overline{\theta}(t)) \\
&= \{(\forall\, x)\, \theta[x \mapsto x]\, F\} \\
&= \{\theta\,(\forall\, x)\, F\} \\
&= \theta\,\{(\forall\, x)\, F\} \\
&= \theta\, FA(D).
\end{aligned}$$

When D is of the form **ex-generalize** $(\exists\, x)\, F$ **from** t, we may assume without loss of generality that

$$x \notin \mathit{Supp}(\theta) \cup \mathit{RanVar}(\theta)$$

so that:

$$\begin{aligned}
FA(\theta\, D) &= FA(\textbf{ex-generalize}\ (\exists\, x)\, \theta\, F\ \textbf{from}\ \overline{\theta}(t)) \\
&= \{\{x \mapsto \overline{\theta}(t)\}\, \theta\, F\} && \text{(Corollary 6.32)} \\
&= \{\theta\,\{x \mapsto t\}\, F \\
&= \theta\,\{\{x \mapsto t\}\, F\} \\
&= \theta\, FA(D).
\end{aligned}$$

When D is a hypothetical deduction **assume** F **in** D', we have:

$$\begin{aligned}
FA(\theta\, D) &= FA(\textbf{assume}\ \theta\, F\ \textbf{in}\ \theta\, D') \\
&= FA(\theta\, D') - \{\theta\, F\} && \text{(inductively)} \\
&= \theta\, FA(D') - \theta\,\{F\}.
\end{aligned}$$

Now $FA(D) = FA(D') - \{F\}$, hence

$$\theta\, FA(D) = \theta\ [FA(D') - \{F\}] \tag{6.144}$$

and thus in order to prove $FA(\theta\, D) = \theta\, FA(D)$ we need to prove

$$\theta\, FA(D') - \theta\,\{F\} = \theta\ [FA(D') - \{F\}].$$

The inclusion

$$\theta\, FA(D') - \theta\, \{F\} \subseteq \theta\, [FA(D') - \{F\}]$$

follows from Lemma 6.58. In the converse direction, we need to show that

$$\theta\, G \in \theta\, FA(D') - \theta\, \{F\} \tag{6.145}$$

for all $G \in FA(D') - \{F\}$. To that end, pick an arbitrary $G \in FA(D')$ such that

$$G \neq F. \tag{6.146}$$

Clearly, $\theta\, G \in \theta\, FA(D')$, so in order to establish 6.145 we only need to show $\theta\, G \neq \theta\, F$. But this follows directly from 6.146 and Lemma 6.63, as w' occurs neither in F nor in G (from Lemma 6.57, since $G \in FA(D')$).

When D is a composition $D_1; D_2$ we have

$$\begin{array}{rcll} FA(\theta\, D) & = & FA(\theta\, D_1; \theta\, D_2) & \\ & = & FA(\theta\, D_1) \cup [FA(\theta\, D_2) - \{\mathcal{C}(\theta\, D_1)\}] & \text{(by induction and Lemma 6.52)} \\ & = & \theta\, FA(D_1) \cup [\theta\, FA(D_2) - \theta\, \{\mathcal{C}(D_1)\}]. & \end{array}$$

Moreover,

$$\begin{array}{rcll} \theta\, FA(D) & = & \theta\, [FA(D_1) \cup (FA(D_2) - \{\mathcal{C}(D_1)\})] & \text{(from Lemma 6.58)} \\ & = & \theta\, FA(D_1) \cup \theta\, [FA(D_2) - \{\mathcal{C}(D_1)\}]. & \end{array}$$

Therefore, to prove $FA(\theta\, D) = \theta\, FA(D)$ we need to prove

$$\theta\, FA(D_2) - \theta\, \{\mathcal{C}(D_1)\} = \theta\, [FA(D_2) - \{\mathcal{C}(D_1)\}].$$

One half of this equality follows from Lemma 6.58. For the other half, we need to show that

$$\theta\, G \in \theta\, FA(D_2) - \theta\, \{\mathcal{C}(D_1)\}$$

for all $G \in FA(D_2) - \{\mathcal{C}(D_1)\}$. To that end, pick any $G \in FA(D_2)$ such that $G \neq \mathcal{C}(D_1)$. Clearly,

$$\theta\, G \in \theta\, FA(D_2)$$

so we only need to show that

$$\theta\, G \neq \theta\, \mathcal{C}(D_1). \tag{6.147}$$

But given that w' occurs neither in G nor in $\mathcal{C}(D_1)$ (Lemma 6.51 and Lemma 6.57), 6.147 follows from Lemma 6.63.

Next, suppose that D is of the form **pick-any** x **in** D'. Either $x = w$ or not. If $x = w$ then

$$\theta\, D = \textbf{pick-any } x \textbf{ in } D' = D$$

hence $FA(\theta\, D) = FA(D)$, while, by Lemma 6.61, $\theta\, FA(D) = FA(D)$, hence the desired equality holds readily in this case. Suppose then that $x \neq w$, so that

$$\theta\, D = \textbf{pick-any } x \textbf{ in } \theta\, D'. \tag{6.148}$$

There are two possibilities:

(1) $FA(\theta\, D') = error$: Inductively,

$$FA(\theta\, D') = \theta\, FA(D')$$

thus we must have $FA(D') = error$. Then, by the definition of FA, $FA(D) = error$ and hence $\theta\, FA(D) = error$. But, by virtue of 6.148, $FA(\theta\, D') = error$ also means that

$$FA(\theta\, D) = error$$

so we have

$$FA(\theta\, D) = \theta\, FA(D) = error.$$

(2) $FA(\theta\, D') \neq error$: Here we distinguish two possible subcases:

(a) $x \in FV(FA(\theta\, D'))$: Inductively,

$$FA(\theta\, D') = \theta\, FA(D')$$

hence $x \in FV(\theta\, FA(D'))$ and, by Lemma 6.60, $x \in FV(FA(D'))$. Therefore, by the definition of FA, we have $FA(D) = error$. From 6.148 and the supposition $x \in FV(FA(\theta\, D'))$ it follows that $FA(\theta\, D) = error$. Hence in this case too we have the identity

$$FA(\theta\, D) = error = \theta\, FA(D).$$

(b) $x \notin FV(FA(\theta\, D'))$: Then by the definition of FA and 6.148, we must have

$$FA(\theta\, D) = FA(\theta\, D'). \tag{6.149}$$

Further, the inductive hypothesis yields

$$FA(\theta\, D') = \theta\, FA(D') \tag{6.150}$$

so the supposition $x \notin FV(FA(\theta\, D'))$ gives

$$x \notin FV(\theta\, FA(D')).$$

Therefore, Lemma 6.60 entails

$$x \notin FV(FA(D')).$$

Accordingly, by the definition of FA we get

$$FA(D') = FA(D) \tag{6.151}$$

and now the desired $FA(\theta\, D) = \theta\, FA(D)$ folows from 6.149, 6.150, and 6.151.

Finally, suppose that D is of the form **pick-witness** x **for** $(\exists\, y)\, F$ **in** D', so that

$$\theta\, D = \textbf{pick-witness}\ x\ \textbf{for}\ \theta\,(\exists\, y)\, F\ \textbf{in}\ \theta[x \mapsto x]\, D'$$

and on the supposition that $y \neq w$ (which can always be ensured by alphabetic renaming), we have:

$$\theta\, D = \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, \theta\, F\ \textbf{in}\ \theta[x \mapsto x]\, D'. \tag{6.152}$$

Now either $x = w$ or $x \neq w$. If $x = w$ then the result is immediate because, since we are assuming that x does not occur free in $(\exists\, y)\, F$, $\theta\, D = D$; while, by Lemma 6.61, $\theta\, FA(D) = FA(D)$, so the identity $FA(\theta\, D) = \theta\, FA(D)$ degenerates into $FA(D) = FA(D)$. Suppose then that $x \neq w$, so that 6.152 becomes

$$\theta\, D = \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, \theta\, F\ \textbf{in}\ \theta\, D'. \tag{6.153}$$

There are two possible cases:

(a) $x \in FV(\mathcal{C}(\theta\, D'))$: Then, by Lemma 6.52, $x \in FV(\theta\, \mathcal{C}(D'))$, and by Lemma 6.60, $x \in FV(\mathcal{C}(D'))$. Therefore, $FA(D) = \mathit{error}$, and thus

$$\theta\, FA(D) = \mathit{error}. \tag{6.154}$$

But, from 6.153, $x \in FV(\mathcal{C}(\theta\, D'))$ also means that

$$FA(\theta\, D) = \mathit{error} \tag{6.155}$$

hence $FA(\theta\, D) = \theta\, FA(D)$ follows from 6.154 and 6.155.

(b) $x \notin FV(\mathcal{C}(\theta\, D'))$: Here we distinguish two subcases:

1. $FA(\theta\, D') = error$: In that case the inductive hypothesis implies

$$FA(\theta\, D') = \theta\, FA(D')$$

thus we must have $FA(D') = error$. Accordingly, $FA(D) = error$ and

$$\theta\, FA(D) = error. \tag{6.156}$$

But $FA(\theta\, D') = error$ also means that

$$FA(\theta\, D) = error \tag{6.157}$$

and hence the desired identity follows from 6.156 and 6.157.

2. $FA(\theta\, D') \neq error$: There are two final subcases here:
   - $x \in FV(FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\})$: Inductively, $FA(\theta\, D') = \theta\, FA(D')$, while θ and $\{y \mapsto x\}$ are disjoint and hence Corollary 6.13 entails

$$\{y \mapsto x\}\, \theta\, F = \theta\, \{y \mapsto x\}\, F.$$

Accordingly, the supposition

$$x \in FV(FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\})$$

becomes

$$x \in FV(\theta\, FA(D') - \theta\, \{\{y \mapsto x\}\, F\}). \tag{6.158}$$

From Lemma 6.58,

$$\theta\, FA(D') - \theta\, \{\{y \mapsto x\}\, F\} \subseteq \theta\, [FA(D') - \{\{y \mapsto x\}\, F\}]$$

and since $\Phi_1 \subseteq \Phi_2$ implies $FV(\Phi_1) \subseteq FV(\Phi_2)$, 6.158 entails

$$x \in FV(\theta\, [FA(D') - \{\{y \mapsto x\}\, F\}]).$$

Therefore, Lemma 6.60 yields

$$x \in FV(FA(D') - \{\{y \mapsto x\}\, F\})$$

and by the definition of FA we have $FA(D) = error$, and thus $\theta\, FA(D) = error$. But the supposition

$$x \in FV(FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\})$$

also means that $FA(\theta\, D) = error$, hence the equality holds in this case too.

– $x \notin FV(FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\})$: First we will show that

$$x \notin FV(FA(D') - \{\{y \mapsto x\}F\}). \tag{6.159}$$

By way of contradiction, suppose that 6.159 does not hold, so that there is a $G \in FA(D')$ such that

$$G \neq \{y \mapsto x\}\, F \tag{6.160}$$

and

$$x \in FV(G). \tag{6.161}$$

Now $G \in FA(D')$ entails $\theta\, G \in \theta\, FA(D')$, and, inductively, $\theta\, FA(D') = FA(\theta\, D')$, so

$$\theta\, G \in FA(\theta\, D'). \tag{6.162}$$

Moreover, we must have

$$\theta\, G \neq \{y \mapsto x\}\, \theta\, F; \tag{6.163}$$

for the supposition $\theta\, G = \{y \mapsto x\}\, \theta\, F$, i.e.,

$$\{w \mapsto w'\}\, G = \{y \mapsto x\}\, \{w \mapsto w'\}\, F \tag{6.164}$$

engenders a contradiction as follows: $\{y \mapsto x\}$ and $\{w \mapsto w'\}$ are disjoint (since $w' \notin \{x, y\}$ from the assumption that w' does not occur in D, and $w \notin \{x, y\}$ from the assumption $w \neq x$[4]), hence

$$\{y \mapsto x\}\, \{w \mapsto w'\}\, F = \{w \mapsto w'\}\, \{y \mapsto x\}\, F$$

and 6.164 becomes

$$\{w \mapsto w'\}\, G = \{w \mapsto w'\}\, \{y \mapsto x\}\, F \tag{6.165}$$

thus

$$\{w' \mapsto w\}\, \{w \mapsto w'\}\, G = \{w' \mapsto w\}\, \{w \mapsto w'\}\, \{y \mapsto x\}\, F \tag{6.166}$$

and since w' does not occur in G (since $G \in FA(D')$) and w' does not occur in $\{y \mapsto x\}\, F$, Corollary 6.18 along with 6.166 imply

$$G = \{y \mapsto x\}\, F$$

[4]While y is a quantified variable that can be assumed to be different from w by renaming.

contradicting 6.160. But 6.162 and 6.163 together entail

$$\theta\, G \in [FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\}] \tag{6.167}$$

while 6.161 and Lemma 6.60 give

$$x \in FV(\theta\, G) \tag{6.168}$$

and finally 6.167 and 6.168 contradict the supposition

$$x \notin FV(FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\}).$$

Thus we conclude $x \notin FV(FA(D') - \{\{y \mapsto x\} F\})$.
Now in light of Lemma 6.52, Lemma 6.60, and the inductive hypothesis, the suppositions $x \notin FV(\mathcal{C}(\theta\, D'))$ and $FA(\theta\, D') \neq \mathit{error}$ entail, respectively, that $x \notin FV(\mathcal{C}(D'))$ and $FA(D') \neq \mathit{error}$. Hence, by the definition of FA, 6.159 entails that

$$FA(D) = [FA(D') - \{\{y \mapsto x\}\, F\}] \cup \{(\exists\, y)\, F\}$$

and since

$$\sigma\, (\Phi_1 \cup \Phi_2) = \sigma\, \Phi_1 \cup \sigma\, \Phi_2$$

for any σ and sets Φ_1, Φ_2, we get

$$\theta\, FA(D) = \theta\, [FA(D') - \{\{y \mapsto x\}\, F\}] \cup \{(\exists\, y)\, \theta\, F\}.$$

Likewise, by our suppositions, we must have

$$FA(\theta\, D) = [FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\}] \cup \{(\exists\, y)\, \theta\, F\}.$$

Therefore, to show $FA(\theta\, D) = \theta\, FA(D)$, we only need to prove

$$FA(\theta\, D') - \{\{y \mapsto x\}\, \theta\, F\} = \theta\, [FA(D') - \{\{y \mapsto x\}\, F\}] \tag{6.169}$$

or, by the inductive hypothesis and the disjointness of θ and $\{y \mapsto x\}$,

$$\theta\, FA(D') - \theta\, \{\{y \mapsto x\}\, F\} = \theta\, [FA(D') - \{\{y \mapsto x\}\, F\}]. \tag{6.170}$$

One inclusion, namely,

$$\theta\, FA(D') - \theta\, \{\{y \mapsto x\}\, F\} \subseteq \theta\, [FA(D') - \{\{y \mapsto x\}\, F\}]$$

follows from Lemma 6.58. In the converse direction, we need to show that

$$\theta\, G \in \theta\, FA(D') - \theta\, \{\{y \mapsto x\}\, F\}$$

for every $G \in FA(D') - \{\{y \mapsto x\}\, F\}$. Accordingly, pick any $G \in FA(D')$ such that

$$G \neq \{y \mapsto x\}\, F. \tag{6.171}$$

Clearly, $\theta\, G \in \theta\, FA(D')$, so we only need to show that

$$\theta\, G \notin \theta\, \{\{y \mapsto x\}\, F\},$$

i.e., that $\theta\, G \neq \theta\, \{y \mapsto x\}\, F$. But this is easily done by contradiction: If $\theta\, G = \theta\, \{y \mapsto x\}\, F$, i.e., if $\{w \mapsto w'\}\, G = \{w \mapsto w'\}\, \{y \mapsto x\}\, F$, then

$$\{w' \mapsto w\}\, \{w \mapsto w'\}\, G = \{w' \mapsto w\}\, \{w \mapsto w'\}\, \{y \mapsto x\}\, F. \tag{6.172}$$

But w' occurs neither in $\{y \mapsto x\}\, F$ (since w' does not occur in D), nor in G (since $G \in FA(D')$, so every variable that occurs in G must also occur in D', and thus in D), hence, by Corollary 6.18, 6.172 becomes

$$G = \{y \mapsto x\}\, F$$

contradicting 6.171.

This completes our case analysis and the inductive argument. ■

Theorem 6.65 $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ *iff* $\beta \supseteq FA(D)$.

Proof: By induction on the structure of D. When D is a claim or a primitive deduction *Prim-Rule* $F_1, \ldots, F_n$, or a universal specialization or existential generalization, the result is readily verified.

Suppose that D is a hypothetical deduction **assume** F **in** D' and that

$$\beta \vdash D \rightsquigarrow \mathcal{C}(D)$$

i.e., $\beta \vdash D \rightsquigarrow F \Rightarrow \mathcal{C}(D')$, so that

$$\beta \cup \{F\} \vdash D' \rightsquigarrow \mathcal{C}(D').$$

Inductively, $\beta \cup \{F\} \supseteq FA(D')$, thus

$$\beta \supseteq FA(D') - \{F\}$$

i.e., $\beta \supseteq FA(D)$. Conversely, suppose that

$$\beta \supseteq FA(D) = FA(D') - \{F\}.$$

Then $\beta \cup \{F\} \supseteq FA(D')$ and the inductive hypothesis yields

$$\beta \cup \{F\} \vdash D' \rightsquigarrow \mathcal{C}(D').$$

Therefore,

$$\beta \vdash \textbf{assume } F \textbf{ in } D' \rightsquigarrow F \Rightarrow \mathcal{C}(D')$$

i.e., $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$.

We continue with compositions. Suppose that D is of the form $D_1; D_2$ and that $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$, so that

$$\beta \vdash D_1 \rightsquigarrow \mathcal{C}(D_1) \tag{6.173}$$

and

$$\beta \cup \{\mathcal{C}(D_1)\} \vdash D_2 \rightsquigarrow \mathcal{C}(D_2) = \mathcal{C}(D). \tag{6.174}$$

Inductively, 6.173 and 6.174 entail

$$\beta \supseteq FA(D_1) \tag{6.175}$$

and

$$\beta \cup \{\mathcal{C}(D_1)\} \supseteq FA(D_2). \tag{6.176}$$

Thus 6.176 gives

$$\beta \supseteq FA(D_2) - \{\mathcal{C}(D_1)\} \tag{6.177}$$

and 6.175 and 6.177 now imply

$$\beta \supseteq FA(D_1) \cup [FA(D_2) - \{\mathcal{C}(D_1)\}]$$

i.e., $\beta \supseteq FA(D)$. Conversely, suppose that $\beta \supseteq FA(D)$, i.e.,

$$\beta \supseteq FA(D_1) \cup [FA(D_2) - \{\mathcal{C}(D_1)\}] \tag{6.178}$$

so that

$$\beta \supseteq FA(D_1) \tag{6.179}$$

and

$$\beta \cup \{\mathcal{C}(D_1)\} \supseteq FA(D_2). \tag{6.180}$$

Inductively, 6.179 and 6.180 entail $\beta \vdash D_1 \rightsquigarrow \mathcal{C}(D_1)$ and

$$\beta \cup \{\mathcal{C}(D_1)\} \vdash D_2 \rightsquigarrow \mathcal{C}(D_2)$$

thus we infer $\beta \vdash D \leadsto \mathcal{C}(D)$.

Next, suppose that D is of the form **pick-any** x **in** D' and that

$$\beta \vdash D \leadsto \mathcal{C}(D)$$

so that

$$\beta \vdash \{x \mapsto z\}\, D' \leadsto \mathcal{C}(\{x \mapsto z\}\, D')$$

for some z that does not occur in β or in D'. By the inductive hypothesis,

$$\beta \supseteq \mathit{FA}(\{x \mapsto z\}\, D'). \tag{6.181}$$

By Lemma 6.64 and our choice of z,

$$\mathit{FA}(\{x \mapsto z\}\, D') = \{x \mapsto z\}\, \mathit{FA}(D') \tag{6.182}$$

hence 6.181 implies

$$\mathit{FA}(D') \neq \mathit{error}. \tag{6.183}$$

In addition, we must have

$$x \notin FV(\mathit{FA}(D')) \tag{6.184}$$

for otherwise Lemma 6.59 would entail that z occurs free in

$$\{x \mapsto z\}\, \mathit{FA}(D') = \mathit{FA}(\{x \mapsto z\}\, D')$$

which is impossible by 6.181 since z does not occur in β. Therefore, by 6.183, 6.184, and the definition of FA, we conclude that $\mathit{FA}(D) \neq \mathit{error}$, and, in particular,

$$\mathit{FA}(D) = \mathit{FA}(D'). \tag{6.185}$$

But 6.184 also means that

$$\{x \mapsto z\}\, \mathit{FA}(D') = \mathit{FA}(D')$$

so by 6.182 we get $\mathit{FA}(\{x \mapsto z\}\, D') = \mathit{FA}(D')$, which, in tandem with 6.181 yields

$$\beta \supseteq \mathit{FA}(D').$$

Finally, by 6.185, this becomes $\beta \supseteq \mathit{FA}(D)$.

Conversely, suppose that

$$\beta \supseteq \mathit{FA}(D). \tag{6.186}$$

This means that $\mathit{FA}(D) \neq \mathit{error}$, and in particular,

$$\mathit{FA}(D) = \mathit{FA}(D') \tag{6.187}$$

and

$$x \notin FV(FA(D')). \tag{6.188}$$

Pick a z that does not occur in β or in D. By 6.188,

$$\{x \mapsto z\}\, FA(D') = FA(D')$$

so by 6.186 and 6.187 we get $\beta \supseteq \{x \mapsto z\}\, FA(D')$. But

$$\{x \mapsto z\}\, FA(D') = FA(\{x \mapsto z\}\, D')$$

thus $\beta \supseteq FA(\{x \mapsto z\}\, D')$. Therefore, by the inductive hypothesis,

$$\beta \vdash \{x \mapsto z\}\, D' \rightsquigarrow \mathcal{C}(\{x \mapsto z\}\, D')$$

and hence, by our choice of z,

$$\beta \vdash \textbf{pick-any } x \textbf{ in } D' \rightsquigarrow (\forall\, z)\, \mathcal{C}(\{x \mapsto z\}\, D')$$

which is to say

$$\beta \vdash D \rightsquigarrow (\forall\, z)\, \mathcal{C}(\{x \mapsto z\}\, D').$$

Now $\mathcal{C}(\{x \mapsto z\}\, D') = \{x \mapsto z\}\, \mathcal{C}(D')$, thus

$$\beta \vdash D \rightsquigarrow (\forall\, z)\, \{x \mapsto z\}\, \mathcal{C}(D'). \tag{6.189}$$

But, by Lemma 6.1,

$$(\forall\, z)\, \{x \mapsto z\}\, \mathcal{C}(D') \approx_\alpha (\forall\, x)\, \mathcal{C}(D')$$

so 6.189 becomes

$$\beta \vdash D \rightsquigarrow (\forall\, x)\, \mathcal{C}(D') = \mathcal{C}(D).$$

Finally, suppose that D is of the form

$$\textbf{pick-witness } x \textbf{ for } (\exists\, y)\, F \textbf{ in } D'$$

and that $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$, so that

$$(\exists\, y)\, F \in \beta \tag{6.190}$$

and

$$\beta \cup \{\{y \mapsto w\}\, F\} \vdash \{x \mapsto w\}\, D' \rightsquigarrow \mathcal{C}(\{x \mapsto w\}\, D') \tag{6.191}$$

for some w that does not occur in β or in D', and where

$$w \notin FV(\mathcal{C}(\{x \mapsto w\}\, D')). \tag{6.192}$$

Inductively, 6.191 gives

$$\beta \cup \{\{y \mapsto w\}\, F\} \supseteq FA(\{x \mapsto w\}\, D') \tag{6.193}$$

so that

$$\beta \supseteq FA(\{x \mapsto w\}\, D') - \{\{y \mapsto w\}\, F\}. \tag{6.194}$$

But

$$FA(\{x \mapsto w\}\, D') = \{x \mapsto w\}\, FA(D') \tag{6.195}$$

and thus 6.193 entails

$$FA(D') \neq \mathit{error}. \tag{6.196}$$

Furthermore, $\mathcal{C}(\{x \mapsto w\}\, D') = \{x \mapsto w\}\, \mathcal{C}(D')$, thus 6.192 means that

$$w \notin FV(\{x \mapsto w\}\, \mathcal{C}(D'))$$

and from Lemma 6.59,

$$x \notin FV(\mathcal{C}(D')). \tag{6.197}$$

We will now show that

$$FA(D) = [FA(D') - \{\{y \mapsto x\}\, F\}] \cup \{(\exists\, y)\, F\}. \tag{6.198}$$

By the definition of FA and in view of 6.197 and 6.196, the only way in which 6.198 could fail to hold is if

$$x \in FV(FA(D') - \{\{y \mapsto x\}\, F\}). \tag{6.199}$$

We will prove that this is not the case via an argument by contradiction. On the supposition that 6.199 holds, there must be a $G \in FA(D')$ such that

$$G \neq \{y \mapsto x\}\, F \tag{6.200}$$

and $x \in FV(G)$. Then

$$\{x \mapsto w\}\, G \in \{x \mapsto w\}\, FA(D')$$

and by 6.195,

$$\{x \mapsto w\}\, G \in FA(\{x \mapsto w\}\, D'). \tag{6.201}$$

Suppose that $\{x \mapsto w\}\, G = \{y \mapsto w\}\, F$. Then

$$\{w \mapsto x\}\, \{x \mapsto w\}\, G = \{w \mapsto x\}\, \{y \mapsto w\}\, F$$

and from Lemma 6.17, $G = \{y \mapsto x\}\, F$, contradicting 6.200. Therefore,

$$\{x \mapsto w\}\, G \neq \{y \mapsto w\}\, F$$

and from 6.201,

$$\{x \mapsto w\}\, G \in FA(\{x \mapsto w\}\, D') - \{\{y \mapsto w\}\, F\}.$$

Hence, from 6.194,

$$\{x \mapsto w\}\, G \in \beta. \tag{6.202}$$

But we have assumed $x \in FV(G)$, thus, by Lemma 6.59, $w \in FV(\{x \mapsto w\}\, G)$, and 6.202 would then mean that $w \in FV(\beta)$, which contradicts the supposition that w does not occur in β. This establishes 6.198. Next, from 6.195, 6.193 becomes

$$\beta \cup \{\{y \mapsto w\}\, F\} \supseteq \{x \mapsto w\}\, FA(D')$$

and therefore, from Lemma 6.58,

$$\{w \mapsto x\}\, \beta \cup \{\{w \mapsto x\}\, \{y \mapsto w\}\, F\} \supseteq \{w \mapsto x\}\, \{x \mapsto w\}\, FA(D')$$

i.e.,

$$\{w \mapsto x\}\, \beta \cup \{\{y \mapsto x\}\, F\} \supseteq FA(D')$$

and thus

$$\{w \mapsto x\}\, \beta \supseteq FA(D') - \{\{y \mapsto x\}\, F\}. \tag{6.203}$$

But w does not occur in β, hence $\{w \mapsto x\}\, \beta = \beta$ and 6.203 becomes

$$\beta \supseteq FA(D') - \{\{y \mapsto x\}\, F\}. \tag{6.204}$$

Finally, by 6.204 and 6.190 we get

$$\beta \supseteq [FA(D') - \{\{y \mapsto x\}\, F\}] \cup \{(\exists\, y)\, F\}$$

which, in view of 6.198, is to say that $\beta \supseteq FA(D)$.

Conversely, suppose that $\beta \supseteq FA(D)$. Then we must have $FA(D) \neq \mathit{error}$, and in particular,

$$\begin{aligned}
x &\notin FV(\mathcal{C}(D')) && (6.205)\\
FA(D') &\neq \mathit{error} && (6.206)\\
x &\notin FV(FA(D') - \{\{y \mapsto x\}\, F\}) && (6.207)\\
\beta &\supseteq [FA(D') - \{\{y \mapsto x\}\, F\}] && (6.208)\\
(\exists\, y)\, F &\in \beta. && (6.209)
\end{aligned}$$

Clearly,

$$[FA(D') - \{\{y \mapsto x\}\, F\}] \cup \{\{y \mapsto x\}\, F\}$$

is a superset of $FA(D')$, hence, by the inductive hypothesis,

$$[FA(D') - \{\{y \mapsto x\} F\}] \cup \{\{y \mapsto x\} F\} \vdash D' \rightsquigarrow \mathcal{C}(D'). \tag{6.210}$$

Pick a z that does not occur in β or in D'. By 6.210 and Theorem 6.33,

$$\{x \mapsto z\} [FA(D') - \{\{y \mapsto x\} F\}] \cup \{x \mapsto z\} \{\{y \mapsto x\} F\} \vdash \{x \mapsto z\} D' \rightsquigarrow \{x \mapsto z\} \mathcal{C}(D'). \tag{6.211}$$

But, by 6.207,

$$\{x \mapsto z\} [FA(D') - \{\{y \mapsto x\} F\}] = FA(D') - \{\{y \mapsto x\} F\}$$

while, by 6.205,

$$\{x \mapsto z\} \mathcal{C}(D') = \mathcal{C}(D')$$

so 6.211 becomes

$$[FA(D') - \{\{y \mapsto x\} F\}] \cup \{\{x \mapsto z\} \{y \mapsto x\} F\} \vdash \{x \mapsto z\} D' \rightsquigarrow \mathcal{C}(D'). \tag{6.212}$$

Since we are assuming that D is normal, $x \notin Var(F)$, hence Lemma 6.17 gives

$$\{x \mapsto z\} \{y \mapsto x\} F = \{y \mapsto z\} F$$

and thus 6.212 becomes

$$[FA(D') - \{\{y \mapsto x\} F\}] \cup \{\{y \mapsto z\} F\} \vdash \{x \mapsto z\} D' \rightsquigarrow \mathcal{C}(D'). \tag{6.213}$$

From 6.208 and the Dilution Theorem we get

$$\beta \cup \{\{y \mapsto z\} F\} \vdash \{x \mapsto z\} D' \rightsquigarrow \mathcal{C}(D'). \tag{6.214}$$

Finally, since z does not occur in D', Lemma 6.51 implies

$$z \notin FV(\mathcal{C}(D')). \tag{6.215}$$

Therefore, in light of 6.214, 6.209, 6.215, and our choice of z, $[R_9]$ yields the desired judgment

$$\beta \vdash D \rightsquigarrow \mathcal{C}(D') = \mathcal{C}(D).$$

This completes the induction. ■

The next corollary follows directly from Theorem 6.65; it is the analogue of Corollary 4.11.

Corollary 6.66 *$Eval(D, \beta) = \mathcal{C}(D)$ iff $FA(D) \subseteq \beta$. Equivalently, $Eval(D, \beta) = error$ iff $FA(D) = error$ or there is some $F \in FA(D)$ such that $F \notin \beta$.*

The next two results are analogues of Lemma 4.19 and Theorem 4.20, respectively, and can be proved with similar reasoning:

Lemma 6.67 *If $D_1 \approx D_2$ then one of two conditions must hold: $\mathcal{C}(D_1) = \mathcal{C}(D_2)$, or*

$$FA(D_1) = FA(D_2) = error.$$

Theorem 6.68 *$D_1 \approx D_2$ iff one of the following holds:*

(a) *$FA(D_1) = FA(D_2) = error$; or*

(b) *$\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $FA(D_1) = FA(D_2)$.*

Accordingly, observational equivalence is decidable.

These definitions and results allow all of the optimization procedures we developed for propositional $\mathcal{CND}$ to carry over without modification; we only need to define their behavior on the new syntactic constructs, which is entirely straightforward in most cases. Recall, for example, that the utility analysis was performed by the following algorithm:

$$
\begin{aligned}
&\mathfrak{U}(\textbf{assume } P \textbf{ in } D) = \textbf{assume } P \textbf{ in } \mathfrak{U}(D)\\
&\mathfrak{U}(D_1; D_2) = \mathit{let}\ \ D_1' = \mathfrak{U}(D_1)\\
&\qquad\qquad\qquad\quad\ \ D_2' = \mathfrak{U}(D_2)\\
&\qquad\qquad\qquad \mathit{in}\\
&\qquad\qquad\qquad\ \ \mathcal{C}(D_1') \notin FA(D_2') \rightarrow D_2', D_1'; D_2'\\
&\mathfrak{U}(D) = D
\end{aligned}
$$

This does not use any concepts other than conclusions and free assumptions. The foregoing results demonstrate that these notions behave in the predicate setting as they did in the propositional case, and thus the correctness proofs for these analyses remain valid.

We close this section with a desugaring of existential specializations in terms of universal generalizations that will prove convenient in Chapter 9. Specifically, suppose we had an additional unary primitive rule **ex-elim** with the following semantics:

$$\emptyset \vdash \textbf{ex-elim } (\forall\, x)\,[F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G] \rightsquigarrow (\forall\, x)\,[F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G]$$
$$\text{provided } x \text{ does not occur free in } G.$$

Thus **ex-elim** has no premises (and is therefore really an axiom), and simply outputs its argument, provided that the latter is of the correct form and the proviso about free occurences of x is observed. We will show in Section 9.7.2 that this is a sound rule by proving that every formula of the form

$$(\forall\, x)\, [F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G]$$

with $x \notin FV(G)$ is a tautology. For now we will simply observe that, with the help of **ex-elim**, we can desugar **pick-witness** deductions into **pick-any** deductions as follows: we view every deduction of the form

$$D = \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, F\ \textbf{in}\ D'$$

as an abbreviation for the following $T(D)$:

1. **pick-any** x **in**
   assume $\{y \mapsto x\}\, F$ **in**
   D';
2. **ex-elim** $(\forall\, y)\, [F \Rightarrow \mathcal{C}(D')] \Rightarrow [(\exists\, y)\, F \Rightarrow \mathcal{C}(D')]$;
3. **modus-ponens** $(\forall\, y)\, [F \Rightarrow \mathcal{C}(D')] \Rightarrow [(\exists\, y)\, F \Rightarrow \mathcal{C}(D')], (\forall\, y)\, [F \Rightarrow \mathcal{C}(D')]$;
4. **modus-ponens** $(\exists\, y)\, F \Rightarrow \mathcal{C}(D'), (\exists\, y)\, F$

The correctness of the desugaring is readily demonstrated:

Lemma 6.69 *Let* $D = \textbf{pick-witness}\ x\ \textbf{for}\ (\exists\, y)\, F\ \textbf{in}\ D'$. *If* $(\exists\, y)\, F \in \beta$ *and*

$$\beta \cup \{\{y \mapsto z\}\, F\} \vdash \{x \mapsto z\}\, D' \rightsquigarrow G$$

for some z *that does not occur in* β *or in* D *or in* $FV(G)$, *then* $\beta \vdash T(D) \rightsquigarrow G$.

Proof: We are assuming that

$$(\exists\, y)\, F \in \beta \tag{6.216}$$

and

$$\beta \cup \{\{y \mapsto z\}\, F\} \vdash \{x \mapsto z\}\, D' \rightsquigarrow G \tag{6.217}$$

for some z that does not occur in β or in D, and where

$$z \notin FV(G). \tag{6.218}$$

Let $w \neq z$ be a variable that does not occur in β or in D or in G. From 6.217 and Theorem 6.33 we get

$$\{z \mapsto w\}\, \beta \cup \{\{z \mapsto w\}\, \{y \mapsto z\}\, F\} \vdash \{z \mapsto w\}\, \{x \mapsto z\}\, D' \rightsquigarrow \{z \mapsto w\}\, G$$

so from our assumptions about z and Lemma 6.17 and Lemma 6.19 we get

$$\beta \cup \{\{y \mapsto w\}\, F\} \vdash \{x \mapsto w\}\, D' \rightsquigarrow G. \tag{6.219}$$

We will refer to the deductions in lines 1–4 of the definition of $T(D)$ as D_1–D_4, respectively. Now since

$$\{x \mapsto w\}\, \{y \mapsto x\}\, F = \{y \mapsto w\}\, F$$

6.219 means that

$$\beta \vdash D_1 \rightsquigarrow (\forall\, w)\, \{y \mapsto w\}\, F \Rightarrow G = \mathcal{C}(\{x \mapsto w\}\, D') \tag{6.220}$$

and since $\mathcal{C}(\{x \mapsto w\}\, D') = \{x \mapsto w\}\, \mathcal{C}(D')$, 6.220 becomes

$$\beta \vdash D_1 \rightsquigarrow (\forall\, w)\ [\{y \mapsto w\}\, F \Rightarrow \{x \mapsto w\}\, \mathcal{C}(D')]. \tag{6.221}$$

But x does not occur free in $\mathcal{C}(D')$, because, from 6.217,

$$G = \mathcal{C}(\{x \mapsto z\}\, D') = \{x \mapsto z\}\, \mathcal{C}(D')$$

so if x occured free in $\mathcal{C}(D')$ then, by Lemma 6.59, z would occur free in

$$\{x \mapsto z\}\, \mathcal{C}(D') = G$$

contradicting 6.218. Thus 6.221 becomes

$$\beta \vdash D_1 \rightsquigarrow (\forall\, w)\ [\{y \mapsto w\}F \ \Rightarrow \mathcal{C}(D')] \tag{6.222}$$

and hence

$$(\forall\, w)\ [\{y \mapsto w\}\, F \Rightarrow \mathcal{C}(D')]$$

and $(\forall\, y)\ [F \Rightarrow \mathcal{C}(D')]$ are alphabetic variants, so 6.222 becomes

$$\beta \vdash D_1 \rightsquigarrow (\forall\, y)\ [F \Rightarrow \mathcal{C}(D')].$$

From this point on the result follows readily. ■

Corollary 6.70 *Let* $D =$ **pick-witness** x **for** $(\exists\, y)\, F$ **in** D' *If*

$$\beta \vdash D \rightsquigarrow G$$

then $\beta \vdash T(D) \rightsquigarrow G$.

Thus we see that in the presence of **ex-elim**, the **pick-witness** construct is superfluous; it is expressible in terms of **pick-any**.

6.5 Tarskian semantics

Structures

Let a logic signature $\mathbf{\Omega} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ be given. An $\mathbf{\Omega}$-*structure* is a pair $\mathcal{D} = (D, \alpha)$ consisting of a non-empty set D, which we call the *domain* (or *carrier*) of $\mathcal{D}$, and a mapping α, called the *realization assignment* of $\mathcal{D}$, that is defined on $\mathcal{C} \cup \mathcal{F} \cup \mathcal{R}$ and is such that

- for every constant symbol $c \in \mathcal{C}$, $\alpha(c)$ is an element of D;
- for every function symbol $f^n \in \mathcal{F}$, $\alpha(f)$ is an n-ary operation on D; and
- for every relation symbol $R^n \in \mathcal{R}$, $\alpha(R)$ is an n-ary relation on D.

$\mathcal{D}$ is also called an *interpretation* of the signature $\mathbf{\Omega}$.

We will write $c^{\mathcal{D}}$, $f^{\mathcal{D}}$, and $R^{\mathcal{D}}$ as abbreviations for $\alpha(c)$, $\alpha(f)$, and $\alpha(R)$, respectively. We call $c^{\mathcal{D}}$, $f^{\mathcal{D}}$, and $R^{\mathcal{D}}$ the *realizations* of the symbols c, f, and R under $\mathcal{D}$, respectively. It is important to bear in mind that c, f and R are *syntactic* objects (symbols), while their realizations $c^{\mathcal{D}}$, $f^{\mathcal{D}}$, and $R^{\mathcal{D}}$ are *semantic* objects (an element of a domain, an operation on that domain, and a relation on that domain, respectively). This distinction can become blurry if the underlying domain D happens to consist of syntactic objects, but it is still useful to think of $\mathbf{\Omega}$ as "syntax" and of an $\mathbf{\Omega}$-structure $\mathcal{D}$ as "semantics". When the signature $\mathbf{\Omega}$ is fixed or immaterial we will simply speak of a "structure" rather than an "$\mathbf{\Omega}$-structure".

As a convention, we will use the same letter for a structure and its domain but in different fonts, e.g., $\mathcal{D}$ for the structure and D for the domain. Thus in any context in which we are speaking of structures $\mathcal{D}$, $\mathcal{K}$, $\mathcal{M}$, …, the letters D, K, M, …, should be understood to signify the respective domains.

Note that an $\mathbf{\Omega}$-structure $\mathcal{D}$ determines a unique $\mathbf{\Omega}$-algebra: the carrier is D and the realizations of c and f are $c^{\mathcal{D}}$ and $f^{\mathcal{D}}$. Indeed, an $\mathbf{\Omega}$-structure is just an $\mathbf{\Omega}$-algebra augmented with realizations for the relation symbols. Accordingly, whenever it is convenient to do so we will ignore the relations and treat an $\mathbf{\Omega}$-structure as an $\mathbf{\Omega}$-algebra.

Finally, if $\mathbf{\Omega}$ is a signature with equality then an $\mathbf{\Omega}$-structure $\mathcal{D}$ will be called *normal* iff the realization of the symbol $=$ is the identity relation on the domain, i.e., iff

$$=^{\mathcal{D}} = \{(d, d) \mid d \in D\}.$$

Clearly, insofar we intend $=$ to signify equality, only normal structures provide the right semantics: if $\mathcal{D}$ is not normal then either we will fail to have $=^{\mathcal{D}}(d, d)$ for some

$d \in D$, or else we will have $=^{\mathcal{D}}(d_1, d_2)$ for two distinct $d_1, d_2 \in D$. From now on we will only be concerned with normal structures, unless we explicity say otherwise.

Valuations and satisfaction

Because an $(\mathbf{\Omega}, \mathcal{V})$-formula might contain free variables, its truth value cannot be determined simply on the basis of an $\mathbf{\Omega}$-structure $\mathcal{D}$. We also need a context that associates variables in $\mathcal{V}$ with elements in the domain D. This is quite analogous to the situation in algebra, where in order to determine the meaning of a term $t \in \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$ we need not only a $\mathbf{\Omega}$-algebra but also a mapping ρ from $\mathcal{V}$ to the algebra's carrier. Accordingly, let $\rho : \mathcal{V} \to D$ be a *valuation*, i.e., a function that assigns an element of the domain D to each variable in $\mathcal{V}$. Bearing in mind that $\mathcal{D}$ is an $\mathbf{\Omega}$-algebra, we write $\overline{\rho}_{\mathcal{D}}$ for the unique homomorphic extension of ρ to $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$, so that

$$\begin{array}{rcl} \overline{\rho}_{\mathcal{D}}(x) & = & \rho(x) \\ \overline{\rho}_{\mathcal{D}}(c) & = & c \\ \overline{\rho}_{\mathcal{D}}(f(t_1, \ldots, t_n)) & = & f^{\mathcal{D}}(\overline{\rho}_{\mathcal{D}}(t_1), \ldots, \overline{\rho}_{\mathcal{D}}(t_n)). \end{array}$$

The existence and uniqueness of $\overline{\rho}_{\mathcal{D}}$ follow from the unique generation of the set of terms over $\mathbf{\Omega}$ and $\mathcal{V}$ [27]. When $\mathcal{D}$ is understood or irrelevant we will simply write $\overline{\rho}$ rather than $\overline{\rho}_{\mathcal{D}}$. The following is a fundamental tool of universal algebra; it is easily proved by a structural induction on t.

Lemma 6.71 (Algebraic Coincidence) *If ρ and σ agree on $Var(t)$ then*

$$\overline{\rho}(t) = \overline{\sigma}(t).$$

Now given an $(\mathbf{\Omega}, \mathcal{V})$-formula F, an $\mathbf{\Omega}$-structure $\mathcal{D}$, and a valuation $\rho : \mathcal{V} \to D$, we define the *satisfaction relation* $\mathcal{D} \models_\rho F$ as follows:

- If F is an atom $R(t_1, \ldots, t_n)$, then $\mathcal{D} \models_\rho F$ iff $R^{\mathcal{D}}(\overline{\rho}_{\mathcal{D}}(t_1), \ldots, \overline{\rho}_{\mathcal{D}}(t_n))$.
- If F is a negation $\neg G$, then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \not\models_\rho G$.
- If F is a conjunction $(F_1 \wedge F_2)$ then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_\rho F_1$ and $\mathcal{D} \models_\rho F_2$.
- If F is a disjunction $(F_1 \vee F_2)$ then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_\rho F_1$ or $\mathcal{D} \models_\rho F_2$.
- If F is a conditional $(F_1 \Rightarrow F_2)$ then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_\rho F_2$ whenever $\mathcal{D} \models_\rho F_1$.

- If F is a biconditional $(F_1 \Leftrightarrow F_2)$ then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_\rho F_2$ whenever $\mathcal{D} \models_\rho F_1$, and $\mathcal{D} \models_\rho F_1$ whenever $\mathcal{D} \models_\rho F_2$.

- If F is a universal quantification $(\forall\, x)\, G$, then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_{\rho[x \,\mapsto\, d]} G$ for every $d \in D$.

- If F is an existential quantification $(\exists\, x)\, G$, then $\mathcal{D} \models_\rho F$ iff $\mathcal{D} \models_{\rho[x \,\mapsto\, d]} G$ for some $d \in D$.

If $\mathcal{D} \models_\rho F$ we say that $\mathcal{D}$ *satisfies* F *relative to* ρ, or that F *holds in* $\mathcal{D}$ *relative to* ρ. Otherwise, if $\mathcal{D} \not\models_\rho F$, we say that $\mathcal{D}$ *falsifies* F *relative to* ρ, or that F *fails in* $\mathcal{D}$ *relative to* ρ. We call F *satisfiable* if there are $\mathcal{D}$ and ρ such that $\mathcal{D} \models_\rho F$, and *unsatisfiable* otherwise. When $\mathcal{D} \models_\rho F$ for *every* valuation ρ, we say that F is *true* (or *valid*) in $\mathcal{D}$, or that $\mathcal{D}$ is a *model* of F. We denote this by writing $\mathcal{D} \models F$.

More generally, for any set $\Phi \subseteq \mathbf{Form}(\boldsymbol{\Omega}, \mathcal{V})$, we write $\mathcal{D} \models_\rho \Phi$ to mean that $\mathcal{D} \models_\rho F$ for every $F \in \Phi$. When $\mathcal{D} \models_\rho \Phi$ we say that $\mathcal{D}$ *satisfies* Φ *relative to* ρ. We call Φ *satisfiable* if $\mathcal{D} \models_\rho \Phi$ for some $\mathcal{D}$ and ρ, and *unsatisfiable* otherwise. If $\mathcal{D} \models F$ for every $F \in \Phi$ we write $\mathcal{D} \models \Phi$ and say that $\mathcal{D}$ is a *model* of Φ. Note that every structure is a model of the empty set, i.e., $\mathcal{D} \models \emptyset$ for every $\mathcal{D}$.

A set $\Phi \subseteq \mathbf{Form}(\boldsymbol{\Omega}, \mathcal{V})$ *logically implies* an $(\boldsymbol{\Omega}, \mathcal{V})$-formula F, written $\Phi \models F$, if for every $\boldsymbol{\Omega}$-structure $\mathcal{D}$ and valuation $\rho : \mathcal{V} \to D$, we have $\mathcal{D} \models_\rho F$ whenever $\mathcal{D} \models_\rho \Phi$. In that case we also say that F is a *logical consequence* of Φ. When Φ is a singleton $\{G\}$ we write $G \models F$ instead of $\{G\} \models F$; and when Φ is the empty set we write $\models F$ instead of $\emptyset \models F$. The reader will verify the following analogue of Lemma 4.25:

Lemma 6.72 *If* $\Phi \cup \{F\} \models G$ *then* $\Phi \models F \Rightarrow G$.

Coincidence and the Substitution Theorem

The following result formally captures the intuition that whether or not $\mathcal{D} \models_\rho F$ holds depends only on the values that ρ assigns to the free variables of F.

Lemma 6.73 (Logic Coincidence) *If* ρ *and* σ *agree on* $FV(F)$ *then*

$$\mathcal{D} \models_\rho F \quad \textit{iff} \quad \mathcal{D} \models_\sigma F.$$

Proof: By structural induction on F. When F is an atom $R(t_1, \ldots, t_n)$, Lemma 6.71 implies

$$\overline{\rho}_{\mathcal{D}}(t_i) = \overline{\sigma}_{\mathcal{D}}(t_i) \tag{6.223}$$

for $i = 1, \ldots, n$, since ρ and σ agree on $Var(t_i)$. Accordingly,

$$\mathcal{D} \models_\rho F \quad \text{iff}$$

$$\begin{array}{ll} R^{\mathcal{D}}(\overline{\rho}_{\mathcal{D}}(t_1),\ldots,\overline{\rho}_{\mathcal{D}}(t_n)) & \text{iff} \quad \text{(from 6.223)} \\ R^{\mathcal{D}}(\overline{\sigma}_{\mathcal{D}}(t_1),\ldots,\overline{\sigma}_{\mathcal{D}}(t_n)) & \text{iff} \\ \mathcal{D} \models_{\sigma} F. & \end{array}$$

The propositional cases are handled by straightforward applications of the inductive hypothesis.

Finally, suppose that F is of the form $(\forall x)\, G$. The key observation is that for any $d \in D$, the valuations $\rho[x \mapsto d]$ and $\sigma[x \mapsto d]$ also agree on $FV(G)$. This allows us to use the inductive hypothesis in the following reasoning:

$$\begin{array}{lll} \mathcal{D} \models_{\rho} F & & \text{iff} \\ \mathcal{D} \models_{\rho[x \mapsto d]} G & \text{for every } d \in D & \text{iff} \quad \text{(from the inductive hypothesis)} \\ \mathcal{D} \models_{\sigma[x \mapsto d]} G & \text{for every } d \in D & \text{iff} \\ \mathcal{D} \models_{\sigma} F. & & \end{array}$$

The case of existential quantifications is similar. ■

A different formulation of this lemma is given below which allows us to handle coincidence of valuations over two different sets of variables. The proof is virtually identical to the one given above.

Lemma 6.74 *Consider an $\mathbf{\Omega}$-structure $\mathcal{D}$ and valuations $\rho : \mathcal{V} \to D$, $\rho' : \mathcal{V}' \to D$, where $\mathcal{V} \subseteq \mathcal{V}'$. If ρ and ρ' agree on $\mathcal{V}$ then for any $F \in$ **Form**$(\mathbf{\Omega}, \mathcal{V})$ we have $\mathcal{D} \models_{\rho} F$ iff $\mathcal{D} \models_{\rho'} F$ (where F is viewed as a member of* **Form**$(\mathbf{\Omega}, \mathcal{V}')$ *in the expression $\mathcal{D} \models_{\rho'} F$).*

Next, consider a vocabulary $(\mathbf{\Omega}, \mathcal{V})$. For any valuation $\rho : \mathcal{V} \to D$ and substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ from $\mathcal{V}$ to **Terms**$(\mathbf{\Omega}, \mathcal{V})$, we define a valuation

$$\rho/_{\mathcal{D}}\, \theta = \rho[x_1 \mapsto \overline{\rho}_{\mathcal{D}}(t_1), \ldots, x_n \mapsto \overline{\rho}_{\mathcal{D}}(t_n)].$$

Thus $\rho/_{\mathcal{D}}\, \theta$ maps a variable x_i to whatever domain element $\overline{\rho}_{\mathcal{D}}$ assigns to the term t_i, and every other variable x to $\rho(x)$. In particular, when t_i is a constant symbol c, $\rho/_{\mathcal{D}}\, \theta$ maps x_i to the individual $c^{\mathcal{D}}$. When $\mathcal{D}$ is fixed or immaterial we might omit it, writing ρ/θ instead of $\rho/_{\mathcal{D}}\, \theta$.

Lemma 6.75 $\overline{\rho/\theta} = \overline{\rho} \cdot \overline{\theta}$.

Proof: We will use structural induction to show that $\overline{\rho/\theta}(t) = \overline{\rho}(\overline{\theta}(t))$ for all t, where $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. If t is a variable x then either $x \in Supp(\theta)$ or not. In the latter case $\overline{\theta}(t) = \theta(x) = x$, thus $\overline{\rho}(\overline{\theta}(t)) = \overline{\rho}(x) = \rho(x)$, while $\overline{\rho/\theta}(t) = \overline{\rho/\theta}(x) = \rho(x)$, hence the equality holds. If $x \in \mathit{Supp}(\theta)$, i.e., if $x = x_i$ for some $i \in \{1, \ldots, n\}$, then

$$\overline{\rho/\theta}(t) = \overline{\rho/\theta}(x_i) = \overline{\rho}(t_i)$$

while

$$\overline{\rho}(\overline{\theta}(t)) = \overline{\rho}(\overline{\theta}(x_i)) = \overline{\rho}(t_i)$$

so the equality holds in this case as well.

For a constant symbol c we have

$$\overline{\rho/\theta}(c) = c^{\mathcal{D}} = \overline{\rho}(c) = \overline{\rho}(\overline{\theta}(c)).$$

Finally, when t is of the form $f(t_1, \ldots, t_n)$ we have

$$\overline{\rho/\theta}(t) = f^{\mathcal{D}}(\overline{\rho/\theta}(t_1), \ldots, \overline{\rho/\theta}(t_n))$$

and

$$\overline{\rho}(\overline{\theta}(t)) = \overline{\rho}(f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n))) = f^{\mathcal{D}}(\overline{\rho}(\overline{\theta}(t_1)), \ldots, \overline{\rho}(\overline{\theta}(t_n)))$$

and the result now follows from the inductive hypothesis. ∎

Lemma 6.76 *If $x \notin \mathit{RanVar}(\theta)$ then $(\rho/\theta)[x \mapsto d] = \rho[x \mapsto d]/\theta[x \mapsto x]$.*

Proof: Let $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. By supposition, $x \notin \mathit{Var}(t_i)$ for $i = 1, \ldots, n$. Pick any variable y and let

$$L(y) = (\rho/\theta)[x \mapsto d](y),$$

$$R(y) = \rho[x \mapsto d]/\theta[x \mapsto x](y).$$

We prove $L(y) = R(y)$ by a case analysis:

(1) $y = x$: In that case $L(y) = d = R(y)$.

(2) $y \neq x$: Here we distinguish two subcases:

(a) $y \in \mathit{Supp}(\theta)$: In that case $L(y) = (\rho/\theta)(y) = \overline{\rho}(t_i)$, while

$$R(y) = \overline{\rho[x \mapsto d]}(t_i).$$

But $x \notin \mathit{Var}(t_i)$ by supposition, hence ρ and $\rho[x \mapsto d]$ agree on $\mathit{Var}(t_i)$, and by Lemma 6.71,

$$\overline{\rho[x \mapsto d]}(t_i) = \overline{\rho}(t_i) = L(y).$$

(b) $y \notin Supp(\theta)$: In that case $L(y) = \rho(y) = R(y)$.

The result now follows since $L(y) = R(y)$ in every possible case. ■

The following is a direct consequence of our definitions:

Lemma 6.77 *If θ_1 and θ_2 agree on $FV(F)$ then so do ρ/θ_1 and ρ/θ_2.*

Theorem 6.78 (Substitution Theorem) $\mathcal{D} \models_{\rho/\theta} F$ *iff* $\mathcal{D} \models_\rho \theta\, F$.

Proof: By induction on F. When F is an atom $R(t_1, \ldots, t_n)$ we have

$$
\begin{array}{lll}
\mathcal{D} \models_\rho \theta\, R(t_1, \ldots, t_n) & \text{iff} & \\
\mathcal{D} \models_\rho R(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_1)) & \text{iff} & \\
R^{\mathcal{D}}(\overline{\rho}(\overline{\theta}(t_1)), \ldots, \overline{\rho}(\overline{\theta}(t_n))) & \text{iff} & \text{(Lemma 6.75)} \\
R^{\mathcal{D}}(\overline{\rho/\theta}(t_1), \ldots, \overline{\rho/\theta}(t_n)) & \text{iff} & \\
\mathcal{D} \models_{\rho/\theta} R(t_1, \ldots, t_n). & &
\end{array}
$$

The propositional cases are handled by straightforward applications of the inductive hypothesis. We illustrate with conjunctions. Suppose F is of the form $F_1 \wedge F_2$. Then

$$
\begin{array}{lll}
\mathcal{D} \models_\rho \theta\, F & \text{iff} & \\
\mathcal{D} \models_\rho \theta\, F_1 \wedge \theta\, F_2 & \text{iff} & \\
\mathcal{D} \models_\rho \theta\, F_1 \text{ and } \mathcal{D} \models_\rho \theta\, F_2 & \text{iff} & \text{(from the inductive hypothesis)} \\
\mathcal{D} \models_{\rho/\theta} F_1 \text{ and } \mathcal{D} \models_{\rho/\theta} F_2 & \text{iff} & \\
\mathcal{D} \models_{\rho/\theta} F_1 \wedge F_2. & &
\end{array}
$$

Finally, suppose that F is a quantified formula of the form $(Q\, x)\, G$. Without loss of generality, we may assume that $x \notin Var(t_i)$, $i = 1, \ldots, n$ (we can always rename α-rename F to ensure this). Now if F is an existential generalization $(\exists\, x)\, G$:

$$
\begin{array}{lll}
\mathcal{D} \models_\rho \theta\, F & \text{iff} & \\
\mathcal{D} \models_\rho (\exists\, x)\, \theta[x \mapsto x]\, G & \text{iff} & \\
\mathcal{D} \models_{\rho[x \mapsto d]} \theta[x \mapsto x]\, G \text{ for some } d & \text{iff} & \text{(from the inductive hypothesis)}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{D} \models_{\rho[x \mapsto d]/\theta[x \mapsto x]} G & \text{iff} & \text{(Lemma 6.76, as } x \notin \mathit{RanVar}(\theta)) \\
\mathcal{D} \models_{(\rho/\theta)[x \mapsto d]} G & \text{iff} & \\
\mathcal{D} \models_{\rho/\theta} (\exists\, x)\, G. & &
\end{array}
$$

Universal generalizations $(\forall\, x)\, G$ are handled similarly:

$$
\begin{array}{lll}
\mathcal{D} \models_{\rho} \theta\, F & \text{iff} & \\
\mathcal{D} \models_{\rho} (\forall\, x)\, \theta[x \mapsto x]\, G & \text{iff} & \\
\mathcal{D} \models_{\rho[x \mapsto d]} \theta[x \mapsto x]\, G \ \text{ for arbitrary } d & \text{iff} & \text{(from the inductive hypothesis)} \\
\mathcal{D} \models_{\rho[x \mapsto d]/\theta[x \mapsto x]} G & \text{iff} & \text{(Lemma 6.76, as } x \notin \mathit{RanVar}(\theta)) \\
\mathcal{D} \models_{(\rho/\theta)[x \mapsto d]} G & \text{iff} & \text{(since } d \text{ is arbitrary)} \\
\mathcal{D} \models_{\rho/\theta} (\forall\, x)\, G. & &
\end{array}
$$

This completes the induction. ■

Lemma 6.79 *If θ_1 and θ_2 agree on $Var(t)$ then $\overline{\theta_1}(t) = \overline{\theta_2}(t)$.*

Proof: A straightforward induction on t. ■

Lemma 6.80 *If θ_1 and θ_2 agree on $FV(F)$ then $\theta_1\, F = \theta_2\, F$.*

Proof: By induction on the structure of F. When F is an atom $R(t_1, \ldots, t_n)$, we have

$$\theta_1\, F = R(\overline{\theta_1}(t_1), \ldots, \overline{\theta_1}(t_n)) \quad \text{and} \quad \theta_2\, F = R(\overline{\theta_2}(t_1), \ldots, \overline{\theta_2}(t_n)).$$

By Lemma 6.79, $\overline{\theta_1}(t_i) = \overline{\theta_2}(t_i)$ for $i = 1, \ldots, n$, hence $\theta_1\, F = \theta_2\, F$. The propositional cases are straightforward with the aid of the inductive hypothesis.

Finally, suppose that F is a quantified formula of the form $(Q\, x)\, G$. We have

$$\theta_1\, F = (Q\, x)\, \theta_1[x \mapsto x]\, G \quad \text{and} \quad \theta_2\, F = (Q\, x)\, \theta_2[x \mapsto x]\, G.$$

It is readily verified that $\theta_1[x \mapsto x]$ and $\theta_2[x \mapsto x]$ agree on $FV(G)$. Hence, by the inductive hypothesis, $\theta_1[x \mapsto x]\, G = \theta_2[v \mapsto v]\, G$ and thus $\theta_1\, F = \theta_2\, F$. ■

6.6 Metatheory

We are now ready to prove the soundness and completeness of first-order $\mathcal{CND}$.

6.6.1 Soundness

We begin by establishing the soundness of the new axioms $[R_6]$ and $[R_7]$, which will be needed in the base case of our inductive argument.

Lemma 6.81 *The universal specialization axiom* $[R_6]$ *is sound. That is,*

$$\beta \cup \{(\forall x)F\} \models \{x \mapsto t\}\, F.$$

Proof: Suppose that for some $\mathcal{D}$ and ρ we have

$$\mathcal{D} \models_\rho \beta \cup \{(\forall x)F\}$$

so that $\mathcal{D} \models_\rho (\forall x)F$. Then, by definition, we must have $\mathcal{D} \models_{\rho[x \mapsto d]} F$ for all $d \in D$, and in particular,

$$\mathcal{D} \models_{\rho[x \mapsto \overline{\rho}(t)]} F.$$

It now follows from the Substitution Theorem that $\mathcal{D} \models_\rho \{x \mapsto t\}\, F$. ■

Lemma 6.82 *The existential generalization axiom* $[R_7]$ *is sound. That is,*

$$\beta \cup \{\{x \mapsto t\}\, F\} \models (\exists x)F.$$

Proof: Suppose that $\mathcal{D} \models_\rho \beta \cup \{\{x \mapsto t\}\, F\}$, so that $\mathcal{D} \models_\rho \{x \mapsto t\}\, F$. By the Substitution Theorem, $\mathcal{D} \models_{\rho[x \mapsto \overline{\rho}(t)]} F$. Accordingly, $\mathcal{D} \models_\rho (\exists x)F$. ■

Lemma 6.83 *The equational axioms* [*Ref*] *and* [*Leibniz*] *are sound.*

Proof: For [*Ref*], we always have $\mathcal{D} \models_\rho t = t$, i.e., $\mathcal{D} \models =^{\mathcal{D}}(\overline{\rho}(t), \overline{\rho}(t))$, since $\mathcal{D}$ is normal. For [*Leibniz*], suppose we have $\mathcal{D} \models_\rho \beta \cup \{s = t\}$, so that $\overline{\rho}(s) = \overline{\rho}(t)$. A straightforward induction on F will show that $\mathcal{D} \models_\rho \{x \mapsto s\}\, F$ iff $\mathcal{D} \models_\rho \{x \mapsto t\}\, F$. ■

Finally:

Theorem 6.84 *If* $\beta \vdash D \rightsquigarrow F$ *then* $\beta \models F$.

Proof: By induction on the structure of D. When D is a claim or a primitive deduction, the supposition that the judgment $\beta \vdash D \leadsto P$ is derivable means that the judgment in question is either an instance of $[R_1]$, or $[R_2]$, or $[R_3]$, or $[R_6]$, or $[R_7]$; or an instance of one of the two equational axioms [*Ref*] and [*Leibniz*]; or an instance of one of the propositional axioms for primitive deductions shown in Figure 4.3. In light of the preceding lemmas for $[R_6]$, $[R_7]$, [*Ref*], and [*Leibniz*], it is readily verified that in any of these cases we have $\beta \models P$.

When D is a hypothetical deduction or a composition, we proceed just as in the propositional case, using Lemma 6.72 (in place of Lemma 4.25) and the transitivity of $\models$. We consider the two remaining cases, universal generalizations and existential instantiations:

- When D is of the form **pick-any** x **in** D', the conclusion F must be of the form $(\forall\, y)\, G$, where we have

  $$\beta \vdash \{x \mapsto y\}\, D' \leadsto G$$

  for some y that does not occur in β or in D'. By the inductive hypothesis, $\beta \models G$. Now suppose that $\mathcal{D} \models_\rho \beta$. Pick any $d \in D$. Since y does not occur free in β, Lemma 6.73 implies $\mathcal{D} \models_{\rho[y \,\mapsto\, d]} \beta$. Thus, since $\beta \models G$, we conclude

  $$\mathcal{D} \models_{\rho[y \,\mapsto\, d]} G.$$

  As this holds for arbitrary $d \in D$, we infer $\mathcal{D} \models_\rho (\forall y) G = F$.

- When D is of the form

  pick-witness x **for** $(\exists\, y)\, G$ **in** D'

  the supposition $\beta \vdash D \leadsto F$ means that we must have $(\exists\, y)\, G \in \beta$ and

  $$\beta \cup \{\{y \mapsto z\}\, G\} \vdash \{x \mapsto z\}\, D' \leadsto F \tag{6.224}$$

  for some z that does not occur in β or in D', and such that $z \notin FV(F)$. Thus from 6.224 and the inductive hypothesis we get

  $$\beta \cup \{\{y \mapsto z\}\, G\} \models F. \tag{6.225}$$

  Now suppose that

  $$\mathcal{D} \models_\rho \beta \tag{6.226}$$

  so that

  $$\mathcal{D} \models_\rho (\exists\, y)\, G \tag{6.227}$$

(as $(\exists\, y)\, G \in \beta$). Since z does not occur in $(\exists\, y)\, G$, we have

$$(\exists\, y)\, G \approx_\alpha (\exists\, z)\, \{y \mapsto z\}\, G$$

so 6.227 yields $\mathcal{D} \models_\rho (\exists\, z)\, \{y \mapsto z\}\, G$, which is to say

$$\mathcal{D} \models_{\rho[z \,\mapsto\, d]} \{y \mapsto z\}\, G \tag{6.228}$$

for some $d \in D$. But z does not occur free in β, so Lemma 6.73 and 6.226 give

$$\mathcal{D} \models_{\rho[z \,\mapsto\, d]} \beta. \tag{6.229}$$

From 6.228, 6.229, and 6.225 we get $\mathcal{D} \models_{\rho[z \,\mapsto\, d]} F$. But $z \notin FV(F)$, hence Lemma 6.73 yields $\mathcal{D} \models_\rho F$. We have thus shown that $\mathcal{D} \models_\rho F$ whenever $\mathcal{D} \models_\rho \beta$, i.e., $\beta \models F$.

The result now follows by structural induction. ∎

We immediately get:

Corollary 6.85 (Soundness) *If* $\Phi \vdash_{\mathcal{CND}} F$ *then* $\Phi \models F$.

6.6.2 Completeness

In this section we will prove that $\mathcal{CND}$ is logically complete, i.e., that $\Phi \vdash_{\mathcal{CND}} F$ whenever $\Phi \models F$. The proof is a standard Henkin argument, but there is one original result, Theorem 6.93, which abstracts aways from different completeness proofs by postulating the existence of a mapping from the underlying domain to the set of terms whose composition with the homomorphic extension of the relevant valuation is the identity on the domain. This allows us, for example, to get completeness as an immediate corollary when we come to cover equality, working with the quotient set of equivalence classes as the underlying domain. There are some other minor technical differences from the standard course, which we feel make for a cleaner proof, e.g., variables are used as witnesses rather than the usual Henkin constants. The reader who is not interested in the details of the proof may take the result on faith and move on.

Preliminaries

We begin with a few definitions and results that carry over directly from the propositional setting. First, we call a set of formulas Φ *inconsistent* if $\Phi \vdash_{\mathcal{CND}}$ **false**, and *consistent* otherwise. Equivalently, Φ is inconsistent iff there is a formula F such that $\Phi \vdash_{\mathcal{CND}} F$ and $\Phi \vdash_{\mathcal{CND}} \neg F$. The proofs of the next two lemmas are as in the propositional case (Lemma 4.29 and Lemma 4.34), since the first-order version of $\vdash_{\mathcal{CND}}$ remains reflexive and monotonic.

Lemma 6.86 *Φ is inconsistent iff $\Phi \vdash_{\mathcal{CND}} F$ for every formula F.*

Lemma 6.87 *$\Phi \vdash_{\mathcal{CND}} F$ iff $\Phi \cup \{\neg F\}$ is inconsistent. Equivalently, $\Phi \vdash_{\mathcal{CND}} \neg F$ iff $\Phi \cup \{F\}$ is inconsistent.*

The definition of maximally consistent sets is the same as before, but with one added subtlety. Specifically, we say that a set $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ is *maximally consistent* iff it is consistent and not properly contained in any consistent set *of* $(\mathbf{\Omega}, \mathcal{V})$*-formulas*. The last qualification is important, as the property is not invariant under vocabulary changes—any consistent set of $(\mathbf{\Omega}, \mathcal{V})$-formulas will remain consistent if we add to it a formula such as $R(v) \vee \neg R(v)$ for some relation symbol R that does not occur in $\mathbf{\Omega}$. Further, we say that $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ is *saturated* iff for any $(\mathbf{\Omega}, \mathcal{V})$-formulas F and G we have

- $F \in \Phi$ iff $\neg F \notin \Phi$. That is, exactly one of the couple $F, \neg F$ is in Φ.
- $F \wedge G \in \Phi$ iff $F \in \Phi$ and $G \in \Phi$.
- $F \vee G \in \Phi$ iff $F \in \Phi$ or $G \in \Phi$.
- $F \Rightarrow G \in \Phi$ iff $G \in \Phi$ whenever $F \in \Phi$.
- $F \Leftrightarrow G \in \Phi$ iff $F \in \Phi$ iff $G \in \Phi$.
- $(\exists\, x)\, F \in \Phi$ iff $\{x \mapsto t\}\, F \in \Phi$ for some t.
- $(\forall\, x)\, F \in \Phi$ iff $\{x \mapsto t\}\, F \in \Phi$ for every t.

The proofs of the next two lemmas also carry over from the propositional setting:

Lemma 6.88 *Maximally consistent sets are deductively closed. That is, if Φ is maximally consistent then $\Phi \vdash_{\mathcal{CND}} F$ iff $F \in \Phi$.*

Note that the converse does not hold—the set of all formulas, for instance, is deductively closed but inconsistent.

Lemma 6.89 (Lindebaum's Lemma) *Every consistent set of $(\mathbf{\Omega}, \mathcal{V})$-formulas is contained in some maximally consistent set of $(\mathbf{\Omega}, \mathcal{V})$-formulas.*

The analogue of Theorem 4.36 does not hold in the first-order setting—it is not true that maximally consistent sets are saturated. However, we will see that maximally consistent sets that *contain witnesses* are saturated.

Henkin extensions and witnesses

Consider any logic vocabulary $(\mathbf{\Omega}, \mathcal{V})$ and let $\mathbf{V}$ be a countably infinite set of variables disjoint from $\mathcal{V}$ and the symbols of $\mathbf{\Omega}$. The vocabulary $(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ is called a *Henkin extension* of $(\mathbf{\Omega}, \mathcal{V})$. The elements of $\mathbf{V}$ are called the *Henkin variables* of $(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$. We will write Henkin variables in boldface, e.g., $\mathbf{v}$, $\mathbf{v}'$, $\mathbf{v}_1$, etc. Clearly, every logic vocabulary has infinitely many Henkin extensions. We note:

(a) $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V}) \subseteq \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$;

(b) $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V}) \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$;

(c) $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ and $\mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ are countable.

Now let us say that a set Φ *contains witnesses* iff for every existential quantification $(\exists\, x)\, F$ in Φ, there is a term t such that $\{x \mapsto t\}\, F \in \Phi$. We have:

Lemma 6.90 *A maximally consistent set that contains witnesses is saturated.*

The proof of the propositional cases carries over directly from propositional logic. The two quantifier cases are straightforward exercises.

The following result is of central importance:

Theorem 6.91 *Consider an arbitrary Henkin extension $(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ of a logic vocabulary $(\mathbf{\Omega}, \mathcal{V})$. Every consistent set $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ is contained in a maximally consistent set $\Phi^* \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ that contains witnesses.*

Proof: Let

$$(\exists\, v_1)\, F_1, (\exists\, v_2)\, F_2, (\exists\, v_3)\, F_3, \ldots \tag{6.230}$$

be an enumeration of all and only the existentially quantified formulas in

$$\mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$$

(such an enumeration exists since there are countably many such formulas). Since $\mathbf{V}$ is countable, there exists a bijection $\psi : N_+ \to \mathbf{V}$. Define an order $\prec$ on $\mathbf{V}$ so that $\mathbf{x} \prec \mathbf{y}$ iff $\psi^{-1}(\mathbf{x}) < \psi^{-1}(\mathbf{y})$. It is easy to check that $\prec$ is a well-ordering. Now define a sequence of Henkin variables $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$ as follows:

$$\mathbf{x}_i = \min_{\prec} [\mathbf{V} - \{\mathbf{x} \mid \mathbf{x} \text{ occurs in } (\exists\, v_j)\, F_j, 1 \leq j \leq i\} \cup \{\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}\}]. \tag{6.231}$$

This is a legitimate definition because $\mathbf{V}$ is infinite, whereas for every $i \in N_+$ the set

$$\{\mathbf{x} \mid \mathbf{x} \text{ occurs in } (\exists\, v_j)\, F_j, 1 \leq j \leq i\} \cup \{\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}\}$$

is finite, so their difference is always non-empty; and since $\prec$ is a well-ordering, every non-empty subset of $\mathbf{V}$ must have a unique $\prec$-least element. Thus, by construction, the following claims hold for all $i \geq 1$:

(1) $\mathbf{x}_i$ does not occur in $(\exists\, v_1)\, F_1, \ldots, (\exists\, v_i)\, F_i$; and

(2) $\mathbf{x}_{i+1}$ is distinct from $\mathbf{x}_1, \ldots, \mathbf{x}_i$.

In what follows we will refer to the $\mathbf{x}_i$ as "witnesses".

For any $i \geq 0$, define

$$\Phi_i = \begin{cases} \Phi & \text{if } i = 0 \\ \Phi_{i-1} \cup \{(\exists\, v_i)\, F_i \Rightarrow \{v_i \mapsto \mathbf{x}_i\}\, F_i\} & \text{if } i > 0. \end{cases} \tag{6.232}$$

By definition, Φ_0 contains no witnesses (since $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$), while for $i \geq 1$, Φ_i does not contain any witnesses outside of $\mathbf{x}_1, \ldots, \mathbf{x}_i$.

An inductive argument on i will show that every Φ_i is consistent. For $i = 0$ this holds by supposition. For $i > 0$ we proceed by way of contradiction. Specifically, suppose that Φ_i is inconsistent. Then, by 6.232 and Lemma 6.87 we conclude

$$\Phi_{i-1} \vdash_{\mathcal{CND}} \neg\, [(\exists\, v_i)\, F_i \Rightarrow \{v_i \mapsto \mathbf{x}_i\}\, F_i]\,.$$

Thus, from Lemma 6.4a,

$$\Phi_{i-1} \vdash_{\mathcal{CND}} (\exists\, v_i)\, F_i \tag{6.233}$$

and

$$\Phi_{i-1} \vdash_{\mathcal{CND}} \neg\{v_i \mapsto \mathbf{x}_i\}\, F_i. \tag{6.234}$$

But $\mathbf{x}_i$ does not occur in Φ_{i-1}, hence, from Corollary 6.36, 6.234 gives

$$\Phi_{i-1} \vdash_{\mathcal{CND}} (\forall\, \mathbf{x}_i)\, \neg\{v_i \mapsto \mathbf{x}_i\}\, F_i$$

and since

$$(\forall\, v_i)\, \neg F_i \approx_\alpha (\forall\, \mathbf{x}_i)\, \neg\{v_i \mapsto \mathbf{x}_i\}\, F_i$$

(as $\mathbf{x}_i$ does not occur in F_i), we conclude $\Phi_{i-1} \vdash_{\mathcal{CND}} (\forall\, v_i)\, \neg F_i$. But this means that $\Phi_{i-1} \vdash_{\mathcal{CND}} \neg(\exists\, v_i)\, F_i$ (Lemma 6.4b), which, combined with 6.233 entails that Φ_{i-1} is inconsistent, contradicting the inductive hypothesis. This completes the inductive argument showing that every Φ_i is consistent.

Now set

$$\Phi' = \bigcup_{i \in N} \Phi_i.$$

By Lemma 6.92, Φ' is consistent. By Lindebaum's Lemma, Φ' is contained in some maximally consistent set $\Phi^* \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$. All that remains to show is that Φ^* contains witnesses.

Suppose that Φ^* contains an existentially quantified formula, which must thus appear in the list 6.230, i.e., it must be of the form $(\exists\, v_i)\, F_i$, for some $i \in N_+$. By construction, Φ^* contains the formula

$$(\exists\, v_i)\, F_i \Rightarrow \{v_i \mapsto \mathbf{x}_i\}\, F_i.$$

Since Φ^* is maximally consistent, it is deductively closed (Lemma 6.88), hence Φ^* contains the formula $\{v_i \mapsto \mathbf{x}_i\}\, F_i$, which proves that it contains witnesses. ∎

Lemma 6.92 *The union of a countable chain of consistent sets is consistent.*

Proof: Let $\Phi_0 \subseteq \Phi_1 \subseteq \Phi_2 \subseteq \cdots$ be the given chain, where each Φ_i is consistent, and set

$$\Phi^* = \bigcup_{i \in N} \Phi_i.$$

By way of contradiction, suppose that Φ^* is inconsistent, so that $\Phi^* \vdash_{\mathcal{CND}}$ **false**, meaning that $\beta \vdash D \rightsquigarrow$ **false** for some D and $\beta \subseteq \Phi^*$. Since β is finite and $\Phi^* = \Phi_0 \cup \Phi_1 \cup \cdots$, we must have $\beta \subseteq \Phi_n$ for some n, hence $\Phi_n \vdash_{\mathcal{CND}}$ **false**, contradicting the assumption that every Φ_i is consistent. ∎

Theorem 6.93 *Let Φ be a maximally consistent set of $(\mathbf{\Omega}, \mathcal{V})$-formulas that contains witnesses, and suppose that there exist an $\mathbf{\Omega}$-structure $\mathcal{D}$, a valuation $\rho : \mathcal{V} \to D$, and a function $\psi : D \to \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$ such that*

(a) *for all n-ary relation symbols R in $\mathbf{\Omega}$ and terms $t_1, \ldots, t_n \in \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$,*

$$\mathcal{D} \models_\rho R(t_1, \ldots, t_n) \quad \textit{iff} \quad R(t_1, \ldots, t_n) \in \Phi;$$

(b) *$\overline{\rho} \cdot \psi$ is the identity on D.*

Then for any $(\mathbf{\Omega}, \mathcal{V})$-formula F, $\mathcal{D} \models_\rho F$ iff $F \in \Phi$.

Proof: By structural induction on F. When F is an atom the result holds by supposition. The propositional cases are readily handled by straightforward applications of the inductive hypothesis and the fact that maximally consistent sets are saturated. We will work out negation as an example and leave the remaining cases as an exercise. When F is a negation $\neg G$ we have:

$$\mathcal{D} \models_\rho \neg G \quad \text{iff}$$

$$\begin{array}{lll} \mathcal{D} \not\models_\rho G & \text{iff} & \text{(by the inductive hypothesis)} \\ G \notin \Phi & \text{iff} & \text{(Lemma 6.90)} \\ \neg G \in \Phi. & & \end{array}$$

Next, suppose that F is of the form $(\exists\, x)\, G$. In one direction, suppose that $F \in \Phi$. Because Φ contains witnesses, we have $\{x \mapsto t\}\, G \in \Phi$ for some term t. From the inductive hypothesis, $\mathcal{D} \models_\rho \{x \mapsto t\}\, G$, so, by the Substitution Theorem,

$$\mathcal{D} \models_{\rho/\{x\,\mapsto\, t\}} G.$$

But $\rho/\{x \mapsto t\} = \rho[x \mapsto \overline{\rho}(t)]$, hence $\mathcal{D} \models_{\rho[x\,\mapsto\,\overline{\rho}(t)]} G$ and thus $\mathcal{D} \models_\rho (\exists\, x)\, G = F$.

Conversely, suppose that $F \notin \Phi$. We will show that

$$\mathcal{D} \not\models_{\rho[x\,\mapsto\, d]} G$$

for all $d \in D$, which will entail $\mathcal{D} \not\models_\rho F$. Pick an arbitrary $d \in D$ and let $t = \psi(d)$. Since $F \notin \Phi$, we have $\Phi \nvdash_{\mathcal{CND}} (\exists\, x)\, G$. Hence $\Phi \nvdash_{\mathcal{CND}} \{x \mapsto t\}\, G$, for otherwise we would have $\Phi \vdash_{\mathcal{CND}} (\exists\, x)\, G$ via **ex-generalize**. Therefore, $\{x \mapsto t\}\, G \notin \Phi$, and the inductive hypothesis implies $\mathcal{D} \not\models_\rho \{x \mapsto t\}\, G$, so the Substitution Theorem gives

$$\mathcal{D} \not\models_{\rho/\{x\,\mapsto\, t\}} G. \tag{6.235}$$

But from the equation $t = \psi(d)$ and our supposition about $\overline{\rho} \cdot \psi$ we get

$$\rho/\{x \mapsto t\} = \rho[x \mapsto \overline{\rho}(t)] = \rho[x \mapsto \overline{\rho}(\psi(d))] = \rho[x \mapsto d] \tag{6.236}$$

hence 6.235 entails $\mathcal{D} \not\models_{\rho[x\,\mapsto\, d]} G$. Since d was chosen arbitrarily, we conclude

$$\mathcal{D} \not\models_\rho (\exists\, x)\, G = F.$$

Finally, suppose that F is of the form $(\forall\, x)\, G$. In one direction, assume $F \in \Phi$. Pick any $d \in D$ and let $t = \psi(d)$. Now $F \in \Phi$ entails $\Phi \vdash_{\mathcal{CND}} F$, hence, using **specialize**, $\Phi \vdash_{\mathcal{CND}} \{x \mapsto t\}\, G$. Thus $\{x \mapsto t\}\, G \in \Phi$, and the inductive hypothesis implies $\mathcal{D} \models_\rho \{x \mapsto t\}\, G$. The Substitution Theorem, in turn, gives $\mathcal{D} \models_{\rho/\{x\,\mapsto\, t\}} G$, and by the reasoning of 6.236 we get

$$\mathcal{D} \models_{\rho[x\,\mapsto\, d]} G.$$

Since this holds for arbitrary d, we conclude $\mathcal{D} \models_\rho (\forall\, x)\, G = F$.

Conversely, suppose that

$$\mathcal{D} \models_\rho (\forall\, x)\, G. \tag{6.237}$$

By way of contradiction, suppose $(\forall\, x)\, G \notin \Phi$, so that $\neg(\forall\, x)\, G \in \Phi$ (since Φ is saturated), and thus $\Phi \vdash_{\mathcal{CND}} \neg(\forall\, x)\, G$. Hence $\Phi \vdash_{\mathcal{CND}} (\exists\, x)\, \neg G$ (Lemma 6.4c), and

$$(\exists\, x)\, \neg G \in \Phi.$$

Since Φ contains witnesses, we have $\neg\{x \mapsto t\}\, G \in \Phi$ for some t, hence

$$\Phi \vdash_{\mathcal{CND}} \neg\{x \mapsto t\}\, G. \tag{6.238}$$

But, from 6.237, $\mathcal{D} \models_{\rho[x \mapsto \overline{\rho}(t)]} G$, and since

$$\rho[x \mapsto \overline{\rho}(t)] = \rho/\{x \mapsto t\}$$

the Substitution Theorem yields $\mathcal{D} \models_{\rho} \{x \mapsto t\}\, G$. Therefore, from the inductive hypothesis, $\{x \mapsto t\}\, G \in \Phi$, and hence $\Phi \vdash_{\mathcal{CND}} \{x \mapsto t\}\, G$, which, in view of 6.238, contradicts the consistency of Φ. Thus we conclude $(\forall\, x)\, G \in \Phi$. ■

All the pieces are now in place for the key result:

Theorem 6.94 *Every consistent set is satisfiable.*

Proof: Let Φ be a consistent set of $(\mathbf{\Omega}, \mathcal{V})$-formulas, and let $(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ be a Henkin extension of $(\mathbf{\Omega}, \mathcal{V})$. By Theorem 6.91, Φ is contained in a maximally consistent

$$\Phi^* \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$$

that contains witnesses. Now define an $\mathbf{\Omega}$-structure $\mathcal{D}$ as follows:

1. the domain D is the set of all terms over $\mathbf{\Omega}$ and $\mathcal{V} \cup \mathbf{V}$, i.e.,
$$D = \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V});$$
2. for any constant symbol c, $c^{\mathcal{D}}$ is the constant symbol c;
3. for any n-ary function symbol f, $f^{\mathcal{D}} : D^n \to D$ is defined as
$$f^{\mathcal{D}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n);$$
and
4. for any n-ary relation symbol R, we set $R^{\mathcal{D}}(t_1, \ldots, t_n)$ iff $R(t_1, \ldots, t_n) \in \Phi^*$.

Next, let $\rho : \mathcal{V} \cup \mathbf{V} \to D$ be the identity mapping on $\mathcal{V} \cup \mathbf{V}$ (we can do this because $\mathcal{V} \cup \mathbf{V} \subseteq D$). It is readily verified that $\overline{\rho}$ is the identity function on D. Now let ϕ be the identity on D as well. It follows that $\overline{\rho} \cdot \phi$ is the identity on D. Hence, by Theorem 6.93, for any $F \in \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ we have $\mathcal{D} \models_{\rho} F$ iff $F \in \Phi^*$. Let $\sigma = \rho \restriction V$, so that σ is the identity on V. Then Lemma 6.74 entails $\mathcal{D} \models_{\sigma} \Phi$, proving that Φ is satisfiable. ■

We conclude:

Theorem 6.95 (Completeness) *If $\Phi \models F$ then $\Phi \vdash_{\mathcal{CND}} F$.*

Proof: If $\Phi \models F$ then $\Phi \cup \{\neg F\}$ is unsatisfiable, hence, by Theorem 6.94, it is inconsistent. Therefore, by Lemma 6.87, $\Phi \vdash_{\mathcal{CND}} F$. ■

Completeness in the presence of equality

It is well-known that the preceding construction will not work for a logic vocabulary with equality because the resulting structure might be non-normal. That is, the realization of the equality symbol will not be the identity relation on the underlying domain—the set of all terms. Suppose, for example, that Φ^* contains the formula $c_1 = c_2$ for two distinct constant symbols c_1 and c_2. Then, by construction, we will have $=^{\mathcal{D}}(c_1, c_2)$. But c_1 and c_2 are two distinct elements of the domain D, hence $=^{\mathcal{D}}$ is not the proper realization of the equality symbol.

However, the construction can be modified in a straightforward way that will result in a normal interpretation. Specifically, let $\Phi \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$ be a consistent set, for some vocabulary $(\mathbf{\Omega}, \mathcal{V})$ with equality. We will construct a normal structure $\mathcal{D}$ that satisfies Φ. The first step is the same as before: we let $(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ be a Henkin extension of $(\mathbf{\Omega}, \mathcal{V})$, and we let $\Phi^* \subseteq \mathbf{Form}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ be a maximally consistent set that contains witnesses. Now we define a binary relation $\equiv$ on $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$ as follows: $s \equiv t$ iff the formula $s = t$ is in Φ^*. We can prove:

Lemma 6.96 *$\equiv$ is an equivalence relation that is compatible with the function and relation symbols of $\mathbf{\Omega}$. That is, for every n-ary function symbol f and n-ary relation symbol R in $\mathbf{\Omega}$, if $s_1 \equiv t_1, \ldots, s_n \equiv t_n$ then*

(a) $f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)$; and

(b) $R(s_1, \ldots, s_n) \in \Phi^*$ iff $R(t_1, \ldots, t_n) \in \Phi^*$.

Proof: We first need to show that $\equiv$ is an equivalence relation: reflexive, symmetric, and transitive. For reflexivity, we have $\Phi^* \vdash_{\mathcal{CND}} t = t$ for arbitrary t (by **ref**), and since Φ^* is deductively closed, $t = t \in \Phi^*$, hence $t \equiv t$. For symmetry, suppose $s \equiv t$, so that $s = t \in \Phi^*$. Then $\Phi^* \vdash_{\mathcal{CND}} s = t$, and from **swap**, $\Phi^* \vdash_{\mathcal{CND}} t = s$, so that $t = s \in \Phi^*$ and hence $t \equiv s$. Transitivity is proved by similar reasoning.

For (a) and (b): if $s_i \equiv t_i$ then $s_i = t_i \in \Phi^*$ and hence

$$\Phi^* \vdash_{\mathcal{CND}} s_i = t_i \tag{6.239}$$

and

$$\Phi^* \vdash_{\mathcal{CND}} t_i = s_i \tag{6.240}$$

for $i = 1, \ldots, n$. Hence, by **cong** and 6.239 we get

$$\Phi^* \vdash_{\mathcal{CND}} f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n).$$

Thus the formula $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ is in Φ^*, and therefore

$$f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n).$$

Likewise, we have

$$\begin{array}{lll} R(s_1, \ldots, s_n) \in \Phi^* & \text{iff} & \\ \Phi^* \vdash_{\mathcal{CND}} R(s_1, \ldots, s_n) & \text{iff} & \text{(by 6.239 or 6.240, and } \textbf{cong}\text{)} \\ \Phi^* \vdash_{\mathcal{CND}} R(t_1, \ldots, t_n) & \text{iff} & \\ R(t_1, \ldots, t_n) \in \Phi^* & & \end{array}$$

which completes the proof of (b). ■

Next we define an $\mathbf{\Omega}$-structure $\mathcal{D}$ as follows. First, we take the domain D to be the quotient class $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})/\equiv$, that is, the set of all equivalence classes determined by $\equiv$. For any constant symbol c, n-ary function symbol f, and n-ary relation symbol R in $\mathbf{\Omega}$, we set

(1) $c^{\mathcal{D}} = [c]$;

(2) $f^{\mathcal{D}}([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$; and

(3) $R^{\mathcal{D}}([t_1], \ldots, [t_n])$ iff $R(t_1, \ldots, t_n) \in \Phi^*$

where we write $[t]$ for the equivalence class of t.

For definitions (2) and (3) to be legitimate we must show that they are independent of how we choose the representatives $t_1, \ldots, t_n$. In particular, suppose that $s_1 \equiv t_1, \ldots, s_n \equiv t_n$ (so that $[s_i] = [t_i]$). For (2), we must show that $[f(s_1, \ldots, s_n)] = [f(t_1, \ldots, t_n)]$, i.e., that $f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)$. But this follows directly from the assumptions $s_i \equiv t_i$ and the preceding lemma. For (3), we must show that

$$R(s_1, \ldots, s_n) \in \Phi^* \text{ iff } R(t_1, \ldots, t_n) \in \Phi^*.$$

This, too, follows directly from $s_i \equiv t_i$ and the previous lemma.

We can now see that $\mathcal{D}$ is normal. For we have

$$\begin{array}{ll} s = t \in \Phi^* & \text{iff} \\ s \equiv t & \text{iff} \\ [s] =^{\mathcal{D}} [t]. & \end{array}$$

That is, $=^{\mathcal{D}}$ relates all and only identical domain elements. Hence $\mathcal{D}$ is normal.

We now define a valuation $\rho : \mathcal{V} \cup \mathbf{V} \to D$ as $\rho(v) = [v]$. A simple structural induction will show that $\overline{\rho_{\mathcal{D}}}(t) = [t]$ for all t. Therefore,

Lemma 6.97 $\overline{\rho_{\mathcal{D}}}(s) = \overline{\rho_{\mathcal{D}}}(t)$ *iff* $s \equiv t$.

We also get:

$$R^{\mathcal{D}}(\overline{\rho_{\mathcal{D}}}(t_1), \ldots, \overline{\rho_{\mathcal{D}}}(t_n)) \text{ iff } R^{\mathcal{D}}([t_1], \ldots, [t_n]) \text{ iff } R(t_1, \ldots, t_n) \in \Phi^*. \tag{6.241}$$

Further, let $\prec$ be some well-ordering on $\mathbf{Terms}(\mathbf{\Omega}, \mathcal{V} \cup \mathbf{V})$, and define $\phi([t])$ as the $\prec$-least element of $[t]$. Thus:

Lemma 6.98 $t \equiv \phi([t])$.

Lemma 6.99 $\overline{\rho_{\mathcal{D}}}(\phi([t])) = [t]$.

Proof: Since $t \equiv \phi([t])$, we have (by Lemma 6.97) that $\overline{\rho_{\mathcal{D}}}(t) = \overline{\rho_{\mathcal{D}}}(\phi([t]))$. But $\overline{\rho_{\mathcal{D}}}(t) = [t]$, hence $\overline{\rho_{\mathcal{D}}}(\phi([t])) = [t]$. ∎

Thus we have shown that $\overline{\rho_{\mathcal{D}}} \cdot \phi$ is the identity on D. In view of 6.241, Theorem 6.93 entails that $\mathcal{D} \models_{\rho} F$ iff $F \in \Phi^*$. It follows from Lemma 6.74 that $\mathcal{D} \models_{\rho \upharpoonright V} G$ for every $G \in \Phi$, and hence that Φ is satisfiable.

7

$\mathcal{CND}$ as a formal analysis of classical reasoning

In this chapter we will consider $\mathcal{CND}$ as a formal explication of deduction and contrast it with previous similar analyses: Hilbert systems, sequent systems, and proof trees.

7.1 Deduction and the nature of formal analysis

The notion of *deduction*, or *proof*, is one of several fundamental intellectual concepts that were around for thousands of years in a fairly well-understood but informal form—in a *pre-theoretical* state. As a methodology for attaining scientific knowledge, and especially for investigating mathematics, it was first introduced in ancient Greece [43]. Two of the most exemplary mathematical proofs of all time (proofs "from the book", to use Paul Erdő's phrase [2]), widely cited in modern textbooks, come verbatim from ancient Greece: the proof that $\sqrt{2}$ is irrational[1] (by the Pythagorians), and the proof

[1]What was actually shown was the incommensurability of the side and the diagonal of a square, but this is just an alternative, geometric formulation of the same statement; the proof proceeds identically regardless of the setting (geometric or numeric). Incidentally, the ancients were severely shocked by the discovery of irrational quantities, and all manners of legend surround the fate of the person responsible for it. According to the historian Proclus: "It is well known that the man who first made public the theory of irrationals perished in a shipwreck, in order that the inexpressible and unimaginable should ever remain veiled."

that there are infinitely many primes (by Euclid). Both are surprisingly modern in their structure, as well as paradigms of insightful, elegant, and succinct deductive reasoning. In his *Elements* Euclid gave 465 rigorous mathematical proofs, starting from five postulates, the first example of systematic axiomatic reasoning in the modern sense. Aristotle developed the subject further in his *Organon*, although many modern logicians tend to reproach his work on the charge that his emphasis on syllogistic reasoning held back the development of predicate logic for two thousand years.[2] Many other proofs were accumulated through the Dark and Middle ages, and even more came after the discovery of analytic geometry, as a result of the general proliferation of mathematical activity that ensued. A good number of the latter, however, especially those dealing with the differentiable calculus, were later found to be flawed; and in fact that was a major motivating factor for the attempt to formalize mathematical reasoning that began with Frege's work.

So although proofs have existed for a very long time, a formal definition of the notion of proof itself was not given until the time of Frege and Russell. Deduction is not alone in this respect. Other fundamental concepts that had been around for a long time but were only defined rigorously in this or the past century include continuity (convergence), the notion of a number (magnitude), and the notion of a mechanical procedure (algorithm). In all of these cases, formal analysis proved to be a tremendously powerful catalyst for scientific progress. The same was to be the case with deduction, but the road here, as we will see, has not been as smooth as in the other cases.

The first formal analysis of deduction, which we will refer to as the Frege-Russell-Hilbert (FRH) analysis, defined a proof as a finite sequence of propositions, each of which is either an axiom or follows from some previous propositions via one of a fixed number of inference rules. Of course this is a formal definition only insofar the notion of proposition is formal. Indeed, casting propositions in symbolic form so as to be able to work with strings of symbols—perfectly sharp mathematical objects—was the key step in the right direction, the one that cleared the path for all subsequent progress; and for this the credit goes directly to Frege and his 1879 *Begriffsschrift*.[3] In our discussion we will take symbolization for granted, since it is entirely unproblematic in relation to our present concerns. And we will do the same for Tarskian semantics, since for our purposes they, too, are non-controversial and routine. Thus we will be

[2] Kant once remarked that Aristotle's work on the subject was so thorough that for all practical purposes Logic was a finished science, complete once and for all.

[3] The importance of Frege's discovery of the variable-quantifier notation that we so readily take for granted today cannot be underestimated. Dummett [24] suggests that Moore's characterization of Russell's theory of descriptions as "a paradigm of philosophy" would have been more aptly ascribed to Frege's discovery.

perfectly happy to treat

$$(\exists x)(\forall y)y \notin x$$

as asserting the existence of an empty set, under the expected *interpretation* of $\in$ and the usual meanings of the logical symbols.

With this background, then, we may pose our analysis problem as follows: *What is a proof?* More specifically, what exactly is a deduction of a proposition P from a set of propositions Γ? We will restrict our attention to first-order reasoning (which is perfectly adequate for mathematics in a set-theoretic universe).

The FRH answer to this question is unsatisfactory because it is not a good model of the informal notion of proof that predated it for thousands of years. Indeed, most readers would be hard-pressed to take either of the two aforementioned ancient proofs, the existence of irrational numbers or the infinitude of prime numbers, and formalize them in FRH style. The reason is that both proofs proceed by way of contradiction, which is a special brand of *hypothetical reasoning*. In hypothetical reasoning one proves P by making some provisional assumption P_1 and then deriving from it some conclusion P_2, the idea being that the derivation of P_2 from P_1 constitutes sufficient justification for inferring P. For instance, the derivation of a contradiction from a hypothesis P entitles us to conclude $\neg P$, and this is precisely what "proofs by contradiction" do, including the two aforementioned proofs by Euclid and the Pythagorians.[4] The second kind of hypothetical reasoning, which in fact subsumes the above,[5] is the method commonly used for establishing conditionals $P \Rightarrow Q$: we take P as a working assumption and attempt to derive Q; if successful, we infer $P \Rightarrow Q$. Hypothetical reasoning permeates mathematical practice, and by far the most serious flaw of the FRH analysis is that it makes no provisions for it.

Note that we are not talking about *extensional accuracy* here. A formal analysis D of an intuitive, pre-theoretical concept C may be viewed as a pair of relations (R, S), where R relates instances of the informal concept C to instances of the formal concept D, while S goes from D to C. The relation R confers extensional ontological status, while S confers intensional status. That is, if $i\,R\,j$ then the definition *proposes*—for analyses are essentially proposals—that we think of j as a formal *hypostasis* ("being") for the (informal) i. And conversely, if $j\,S\,i$ then the definition proposes that we intuitively understand j as the informal object i. For instance, in Turing's analysis of computability, R might map Euclid's algorithm (an informal object) to some Turing machine that carries it out; while S maps Turing machines to the informal algorithms

[4] This technique of "indirect proof" is essentially the Socratic method of dialog, which aims to discredit an assertion by showing that it engenders a contradiction. Plato frequently illustrated this method (also known by the Latin term *reductio ad absurdum*) by citing the proof that the diagonal of the unit square is of irrational length.

[5] See our desugaring of **suppose-absurd** in terms of **assume** in Section 4.1.

that they embody. Now a formalization is *extensionally accurate* if for *every* instance i of the informal concept C there is *some* instance j of D such that $i\,R\,j$; and conversely, if for every instance j of D there is some instance i of C such that $j\,S\,i$. This second half is usually the easy part. Take any Turing machine, for instance, no matter how large or bizarre or senseless. We can, with a modicum of good will, view such a machine as embodying an informal algorithm, albeit a very strange or useless algorithm. Or consider even the formalization of propositions as "well-formed formulas", as another example. Any given wff, even silly ones such as $\mathbf{true} \vee (\forall x)\mathbf{true}$, can still be viewed as expressing a proposition, albeit a silly one. A useless algorithm, after all, is an algorithm nevertheless; and a silly proposition is still a proposition.

The other half, however, is trickier. In fact the extensional accuracy of a formalization can never be rigorously established, since the principle of extensional accuracy is a universally quantified *empirical* proposition ("for every instance i of the *informal* concept C, ..."), akin to empirical laws such as "All whales are white". Such propositions can never be conclusively established, although they can be conclusively falsified (as the above law was indeed falsified when the Europeans discovered black whales in Australian waters). That is why Church's thesis, which is the assertion that Turing's analysis is extensionally accurate, can never be mathematically proven. There will always remain the possibility, however slim, that some informal algorithm manages to do something that no Turing machine can. Nevertheless, when a formalization has withstood the test of time, meaning that no counterexamples have been discovered despite extensive research and experience; when it has been found co-extensive with all other sensible formalizations of the same concept; and when it has enabled us to derive results that make intuitive sense; then we are perfectly justified to accept the formalization as extensionally accurate, even if we must do so on a thereotically tentative basis. That is why Church's thesis, for instance, is widely accepted.

Now in the case of deduction it transpired, thanks to Gödel's work, that the FRH analysis captured a provability relation that could be extensionally characterized independently of any particular formalization of the notion of proof, and which furthermore was intuively deemed to be just the right extension for that concept: namely, P is provable from Γ iff P is logically entailed by Γ. The latter condition is purely semantic, and makes no reference to any particular formal proof system. This extensional identification is the famous logical completeness result for first-order FRH proof systems.[6] Later

[6]The fact that systems so stark as FRH calculi would turn out logically complete might appear combinatorially striking at first, but nowadays it is no surprise: all one needs to do is choose the axioms and the rules so as to make inter-provability of two propositions an equivalence relation, taking care that the resulting quotient class should become a Boolean algebra when one defines a partial order $|P| \leq |Q|$ as Q being derivable from P; the ultrafilter theorem then takes care of completeness automatically. Henkin witness methods have likewise been streamlined for the first-order case.

on other formal analyses of the same concept (e.g., Gentzen's systems), also turned out to be logically complete, and hence co-extensive with the FRH analysis.

In view of this it becomes eminently reasonable to take logical entailment as the touchtone of extensional accuracy for any formal analysis of (classical first-order[7]) proof. That is, such an analysis is deemed extensionally correct iff a formal proof that derives P from Γ exists iff Γ entails P (this is usually expressed symbolically by writing $\Gamma \models P$).

So when we are saying that the FRH analysis is unsatisfactory, we are not referring to its extensional accuracy. In general, extensional accuracy is not the only standard by which a formalization ought to be judged. It is, of course, a necessary condition for a good analysis, but it is not quite sufficient. Two other important classes of criteria are the following:

- **Intensional accuracy**: When the analysis relates an informal object i to a formal one j, i.e., when we have $i\,R\,j$, how natural is it for us to think of j as a formal counterpart of i? Is there an evident conceptual resemblance, or is the correspondence *ad hoc*, to the point where we really have to strain our imagination to "see" it?[8] As an example, consider three different formalizations of computability:

    1. a machine language (with programs being sequences of 0s and 1s, to be interpreted according to the rules of some Turing-complete abstract machine);
    2. Turing machines; and
    3. your favorite modern high-level language.

    Now take Euclid's algorithm as the informal object i, and consider three formal objects j_M (a machine-language program), j_T (a Turing machine), and j_H (a program in the high-level language), each of which carries out Euclid's algorithm in its respective formalism. How natural is it for us to think of j_M as Euclid's algorithm? Is there any conceptual resemblance between the two? How about j_T? And j_H? Now all three formalizations are extensionally identical, but intensionally it is apparent that they are not all of the same merit.

    A good acid test here is this: find someone who is familiar with the informal concept, present the formal analysis to them, give them arbitrary informal i, and

[7] Intuitionist analyses, for example, are not extensionally complete by choice, as they reject classic Tarskian semantics. But they have an agenda—they are not formal analyses of what mathematical reasoning actually *is*, but rather of what it *should be*.

[8] These are of course psychological considerations, not mathematical, but keep in mind that formal analyses are essentially proposals, intent on shedding light on some hitherto vague notion; their very raison d'être is explication, a largely psychological notion.

ask them to find formal counterparts of i, for several different i. This can yield a relative quantitative measure of the foregoing attributes: If we have two different but co-extensive analyses of the same concept, and test performance is consistently better for one of them, with a difference that is statistically significant, then that analysis is surely to be preferred.

- **Pragmatic considerations**: What is the theoretical muscle of the analysis? Does the analysis lead to reasonable, intuitive results? Does it allow us to formalize other related notions? More importantly, how *easy* is it to derive "the right results"? Could our task be easier if we adopted some other analysis?

  Pragmatic considerations can sometimes be at odds with intensional accuracy. For instance, Zermelo's original definition of the natural numbers as

  $$0 \equiv \emptyset, 1 \equiv \{\emptyset\}, 2 \equiv \{\{\emptyset\}\}, \ldots$$

  is a perfectly sensible formalization of the concept of number. Contrast it with the now-customary definition

  $$0 \equiv \emptyset, n+1 = \{n\} \cup n$$

  (due to Mirimanoff, although commonly attributed to Von Neumann). Now, intensionally, Zermelo's definition is better. It's much easier for me to conceive and express the number six in his formalization than in Mirimanoff's, since transitive well-ordered sets, elegant as they may be, are not particularly intuitive. Yet Mirimanoff's definition is preferred because it affords a smooth generalization of the ordinal notion to the transfinite case; and this is extremely important in set theory.

  However, note that insofar the theorems we derive are to be interpreted as results about the pre-theoretical, informal objects, intensional accuracy is extremely important; for if the formal objects are not good models of their informal counterparts, in the intensional sense, then their properties will not necessarily be reflected by those counterparts. Consequently, deriving such properties, although perhaps interesting and challenging as a research project in and of itself, will not necessarily tell us something about the informal concept, which is really what we wanted to elucidate in the first place. Consider, for instance, the formalization of natural deduction by proof trees in which disharged assumptions are tagged by labels attached to nodes of the tree (a typical system of this kind can be found in Section 1.3.1 of "Basic Proof Theory" by Troelstra and Schwichtenberg [68]; we will discuss such systems in more detail shortly). In fact there is nothing "natural" about this kind of deductive system, and as an analysis of the notion of proof it is patently unsatisfactory. Nevertheless, it might be of interest to study such proof trees as objects of investigation in and of themselves. But to call that "proof theory" is presumptuous at best and incorrect at worst, for it tacitly presupposes that the formal objects under investigation merit identification with "proofs" in the informal sense. That is, it presupposes the acceptance of the proposal behind the underlying formal analysis, the proposal of course being "We propose to think of a proof as a tree such that ...". But we will see that there are in fact many good reasons to reject that analysis. So strictly speaking what one should call this subject is "theory of proof trees", or something to that effect. The point is that one cannot simply impose the results obtained by a formalization (tagged proof trees) on the informal objects (proofs) without evidence that the formalization is intensionally accurate.[9]

These are simply two very general classes of criteria that are desirable in most formalizations; they are not an exhaustive enumeration of requirements, nor do they provide a mechanical guide for evaluating formalizations. Some formalizations have well-defined, specific goals, and their success can only be measured in accordance with the extent to which they meet those goals. For example, a formalization of "proof" might be given especially so that it facilitates mechanical proof search, the goal being a formal system in which proofs can be efficiently discovered by an algorithm. Genzten's LK and LJ calculi [29], for instance, should be viewed in this light, as should be Beth's semantic tableau [7, 6]. In such cases one does not make any metaphysical or epistemological claims, such as "our formal analysis elucidates the notion of proof"; all that is made is an attempt to achieve a particular stated objective.

We will summarize the foregoing discussion with a reference to the first chapter of Carnap's "Logical foundations of probability" [13], which is also devoted to a discussion of the general principles of formal analysis (he uses the term "explication" instead of "analysis"). Carnap singles out three criteria[10] for assessing a formal concept that is being proposed as an analysis of an informal one:

Similarity The formal concept should be similar to the informal notion it is intended to replace, "in such a way that in most cases in which [the informal concept] has been used, [the formal concept] can be used".

[9] Of course in practice the use of the term "proof theory" is completely innocuous, serving simply as a convention for labelling a particular field of study as distinct from other related disciplines, and as such it is perfectly reasonable. And in fact it is not specific to any one formalization of deduction; it studies a fairly wide variety of systems. I am only using it as an example to call attention to a general point about formalizations.

[10] In fact he singles out four criteria, but the fourth one, that of "exactness", is extraneous in our case because we are only concerned with formal explications, where the explicatum is an exact (formal) concept by supposition.

Fruitfulness The formal concept should be fruitful, meaning that it should be conducive to the development of a rich theory, allowing for the formulation of many universal statements ("empirical laws in the case of a nonlogical concept, logical theorems in the case of a logical concept").

Simplicity The formal concept should be as simple as possible; "this means as simple as the more important requirements [of similarity and fruitfulness] permit".

Note that similarity falls under what I have dubbed "intensional accuracy", whereas fruitfulness and simplicity are what I have called "pragmatic considerations".

Now the principal claim of this chapter is that, as a formal analysis of deduction, $\mathcal{CND}$ fares better than its predecessors:

- $\mathcal{CND}$ proofs are more similar to informal proofs—they preserve their structure much better thanks to composition and assumption/eigenvariable scope;
- they are "fruitful" in Carnap's sense because they are remarkably amenable to rigorous analysis and give rise to a rich body of theory (proof equivalence, optimization, etc.); and
- they are simple—this is what we mean by "readability" and "writability". Reading and writing $\mathcal{CND}$ proofs is, in our experience, considerably easier than in other systems.

To support these claims comparatively, we will critically examine the three most prominent formalizations of deduction: FRH systems, sequent systems, and proof trees. Although our focus is on classical logic, most of what we have to say will apply to other logics as well, since the weaknesses to which we will call attention carry over unchanged when these systems are adapted to intuitionist logics, higher-order logics, and so on.

7.2 A critique of previous formalizations of deduction

7.2.1 FRH systems

FRH systems are very poor analyses of deduction. They are extensionally correct; and it is not difficult to reason about them because of their simple linear structure. But anyone who is familiar with them will attest that, as models of actual proofs, they are horrific. Wilfrid Hodges [39] has called them "barbarously unintuitive", and that is one of the more polite epithets that can be ascribed to them. Constructing proofs in such systems is extremely impractical (this is "the acid test" we mentioned above in the discussion of intensional accuracy). The key problem is that they do not accomodate hypothetical reasoning. As a result, informal proofs of conditionals and negations (and most interesting resuls in mathematics are conditionals and negations, not conjunctions or disjunctions), are impossible to formalize. Roundabout ways must be found. People who persist working in a FRH system eventually drift into doing proofs at a meta level ("such and such *can be derived* from such and such, therefore *there exists* a derivation of P from Γ), invoking meta-theoretic results such as the "deduction theorem". At that point they are essentially working in a Gentzen-type system, to be discussed below, which can be seen as meta versions of FRH systems.

Another defect of FRH systems, which is actually shared by all other common formalizations, is that the semantics of a proof are baked into its very definition: a proof is *defined* not just as any sequence of formulas, but as a sequence of formulas each of which is either an axiom or properly follows from previous formulas through some sound inference rule. This entails the paradoxical conclusion that all proofs are sound—*ex cathedra*. This is as realistic as the assertion that all computer programs are correct. The opposite conclusion is borne out by a brief inspection of the history of mathematics, or simply by grading homeworks for a math or logic class. Some proofs contain subtle errors, other make patent mistakes, others are trivially sound, some are sound *relative to a given set of assumptions* but may become incorrect if some of those assumptions are retracted, others are incorrect relative to a set of assumptions but may become correct if some additional assumptions are postulated, and so on. It is the purpose of proof-checking to separate the sound proofs from the flawed ones. Of course this is not a deep difference; it is largely a matter of viewpoint.[11] Proof-checking in a FRH system, as well as in every other system, amounts to type-checking: we simply check to see whether a given sequence of formulas actually *is* a proof. But it is a more appropriate viewpoint to regard a proof as correct relative to a given specification: does it derive a given P from a given Γ? DPLs adopt this viewpoint by enforcing a sharp distinction between the syntax of deductions and their semantics.

7.2.2 Sequent systems

It was dissatisfaction with the FRH analysis that prompted Gentzen to seek a different formalization, one that would better capture what working mathematicians, logicians, and philosophers had traditionally understood by the term "deduction". (Gentzen

[11] It is similar to the difference between defining terms in a typed language with a context-free grammar and then introducing a type system to weed out the well-typed terms from the ill-typed ones, versus defining terms with a context-sensitive grammar that incorporates the typing rules, so that all terms are by definition well-typed.

was in fact predated by Łukasiewicz, who was the first logician to openly criticize FRH systems and stress the need for a different formalization. The credit for "natural deduction" should properly go to the Polish school—Łukasiewicz and Jáskowski in particular; we will discuss their contributions further later on. Gentzen's work, however, was independent of the Polish developments.) He correctly perceived that the main problem of FRH systems is their inability to accomodate hypothetical reasoning. He tackled the problem by using *sequents*: pairs of the form (Γ, P).[12] The propositions in Γ represent the assumptions from which P is derived—the "assumption base", in DPL terminology. Formally, the meaning of a sequent $(\{P_1, \ldots, P_k\}, P)$ can be understood as the assertion that the set $\{P_1, \ldots, P_k\}$ logically entails P, so that the sequent is tantamount to the conditional $P_1 \wedge \cdots \wedge P_k \Rightarrow P$. But informally one might think of it as the assertion that P is derivable from Γ in some FRH system. Then rules such as "from $(\Gamma, A \wedge B)$ infer (Γ, A)" can be understood as statements about the meta-theory of a FRH system: if $A \wedge B$ can be deduced from Γ, then so can A. This interpretation is more faithful to Gentzen's original motivations (e.g., see Kleene's "Mathematical Logic" [42], Chapter 1, Section 11). The rule for introducing conditionals, in particular (7.1 below), embodies the meta-theoretic result known as "the deduction theorem" for FRH systems.

Now in such a system a proof is defined as a sequence of sequents, each of which is either an "axiom" sequent (i.e., a sequent whose validity is obvious, such as $(\Gamma \cup \{P\}, P)$), or else follows from some previous sequents through the application of some inference rule such as the one mentioned in the last paragraph, or such as

$$\text{From } (\Gamma \cup \{P\}, Q) \text{ infer } (\Gamma, P \Rightarrow Q) \tag{7.1}$$

or

$$\text{From } (\Gamma, P) \text{ and } (\Gamma, Q) \text{ infer } (\Gamma, P \wedge Q).$$

(It is also possible to define a proof as a tree of sequents, where every leaf is an axiom sequent and every internal sequent is obtainable from its children through a rule, but

[12] Gentzen also formulated systems using sequents of the form (Γ_1, Γ_2), understood as asserting that Γ_1 entails at least one proposition in Γ_2. Such systems, however, are not so much concerned with modeling the informal notion of proof as they are with automatic proof search. That is, they are not meant as frameworks in which people can formally express a proof that they already have in their mind informally, but rather as frameworks in which one would search for a proof of a result that they want to establish. Differently put, they are *analytic* (or *backwards*) systems, rather than *synthetic* (or *forward*) systems. The idea is that the system will try to decompose a given "goal" into a number of simpler subgoals (hence the term "analytic"), and so on with each subgoal, until all the subgoals are trivial. These systems lend themselves to this approach because, unlike forward systems, they have the so-called "subformula property". Of course such systems can also be used in a forward manner, and although I doubt that anyone would seriously advocate this for the task of expressing proofs, all the criticisms we level against sequent systems of the "natural deduction" variety (based on sequents of the form (Γ, P)) will also apply to these.

this is largely a stylistic variation; all of the criticisms we put forth below will also apply to tree versions.)

How good a model of deduction is this? It represents an improvement over the FRH analysis, because at least it begins to address the issues of assumption scope and hypothetical reasoning; but it does not resolve these issues in a satisfactory manner. Although it is certainly easier to express a proof in a sequent system than in a FRH system, the approach is marred by serious problems. Proofs in the real world manipulate individual propositions, not sequents. Euclid did not prove that there are infinitely many primes by pushing sequents around. Pick any mathematics or science or philosophy journal or textbook, or indeed even a newspaper describing a logical argument. People deduce propositions, not sequents. And in the process, they apply inference rules such as Modus Ponens to single propositions, *not* to pairs consisting of an assumption base and a proposition. That is, in the course of a typical proof one applies Modus Ponens to two propositions of the form $P \Rightarrow Q$ and P, which are already known to hold, to obtain Q. That seems crystal clear. But in a sequent system the same step is considerably more complicated, as Modus Ponens needs to be applied to two *sequents* (Γ_1, P_1) and (Γ_2, P_2) that have already been derived, in order to derive another sequent (Γ_3, P_3). This is unsatisfactory in several respects:

- It is not a good model of informal proofs .

- It is mentally taxing because the arguments to the rule, as well as the result, are substantially more complex than they are in practice. One needs to keep track of three *sets* of propositions (Γ_1, Γ_2, and Γ_3), each of which might well contain dozens of propositions, *as well* as the three individual propositions (P_1, P_2, and P_3), *as well* as the relations between the three sets and the three propositions (namely, that S_1 is paired up with P_1, S_2 with P_2, etc.). By contrast, in practice only P_1, P_2, and P_3 are relevant.

- It obfuscates the structure of the proof, because the presence of the contexts S_i clutters up the picture. In fact the presence of the S_i is gratuitous in connection with the application of Modus Ponens, as the latter is commonly understood.

- It is inefficient, since a lot more data needs to be maintained and manipulated with every application of an inference rule.

In Carnap's terminology, sequent proofs are not similar to informal proofs; and they are not simple. They are also more difficult to analyze than $\mathcal{CND}$ proofs, because the scopes of the various hypotheses are not as readily identifiable.

The gist of the matter is that people do not bother to carry around every single assumption that they are making at every step of the way. And this is just as well, for there is no reason why they should; to do so would be superfluous. They only mention the assumptions that they actually need, and then *only* when they actually use them. An analogy with programming might illuminate the situation. Consider an assignment command such as `x := y + 1`, in some modern imperative programming language. The effect of this command is to store the value of the expression `y + 1` in the location denoted by `x` *in the current store*. What does "the current store" mean? It has to do with the semantics of the language. If you look at the formal denotational semantics of the language you will see that every command is executed *relative to a given store*. The semantics themselves specify, in perfect detail, how a given command interacts with a given store. Then an abstract machine that implements those semantics starts executing a program in some initial store and moves through the various commands appropriately, automatically updating the store to account for the effect that each command has, as prescribed by the semantics. Thus the handling of the store is streamlined *by the semantics of the language*, which automatically ensure that the store is properly threaded from each command to the next.

Now imagine a programming language in which an assignment command must take a store explicitly as as an argument. In such a language you would not be able to simply say something like `x := y + 1`. Instead, you would have to supply each assignment statement with *three* arguments: a store S, a variable v (the target of the assignment), and an expression e (the value of which will be stored in the location denoted by v). The result of the assignment command would be the store S' obtained from S by storing in location v the value obtained by evaluating e in S. The purpose of a program, then, would be to start at some initial state (some "initial sequent", in the case of sequent proofs) and eventually produce the right store S' ("right" in that it contains the desired computational results in some particular locations). Programming in such a language would be highly impractical. The language would suffer from exactly the four defects that we mentioned above in connection with sequents: it would not reflect the way in which imperative computations are described in practice; it would be mentally taxing, because each command would take and produce stores as arguments in addition to whatever other parameters it would normally take; it would obfuscate the structure of the algorithm; and it would be inefficient.[13]

That is why in practice the programmer does not need to manipulate stores explicitly. Instead, store manipulation is *implicit*, baked into the semantics of the language at a foundational level, and thus automatically performed at run time. This relieves the programmer of a great burden resulting in shorter, more readable, and cleaner code. Likewise, we claim that a proof engineer should not have to explicitly manipulate as-

[13] In fact that is how state can be theoretically simulated in purely functional languages. It is well-known that for most applications with a strong imperative component (dealing with mutable objects such as bank accounts), such a simulation would be unduly cumbersome [30, 1].

sumption bases, as sequent systems require. Incorporating the assumption base into the underlying semantics streamlines its handling and relieves the user of that concern, resulting in shorter, more readable, and cleaner proofs, that more closely resemble the informal arguments one encounters in practice. This is perhaps the principal tenet of the DPL approach.

One might think that making the assumption base explicit is necessary for dealing with assumption introduction and discharge in hypothetical reasoning, but that is not the case. In actual proofs, when we introduce an assumption P it is tacitly understood that the assumption base, *whatever it might be at that point*, will be augmented by P; there is no reason to mention its actual contents. Furthermore, and this is a crucial observation, *the scope of an assumption is emergent from the lexical structure of the proof.* Accordingly, assumptions do not need to be explicitly discharged, for it is clear just by visual inspection of the text of the proof exactly when a particular thread of hypothetical reasoning is complete, and hence exactly when the corresponding hypothesis ceases to be in effect. We quote De Brujin [11], speaking about "what is sometimes called the Fitch style of natural deduction" (emphasis mine):

> It fully deserves to be called natural, since it was used by mathematicians long before it was ever formally described: namely, the presentation of mathematics in the form of nested blocks, where blocks are opened either by making an assumption or by introducing a (possibly typed) variable (*but it should be admitted that the action of closing a block was usually not mentioned explicitly, just suggested by the subdivision of texts into sentences, paragraphs, sections, etc.*).

This correspondence cannot be captured in sequent systems because the scope of a hypothesis, as we understand it in a regular proof, loses its meaning in a sequent proof. In a regular proof, the scope of an assumption is the portion of the proof within which the assumption remains active and may be freely asserted. But in a sequent proof the scope of an assumption makes sense only with respect to a single sequent, not with respect to the entire proof, because the entire sequent proof itself is linear—it has no nested structure, and hence no notion of scope. Thus important notions such as that of a *subdeduction*, which naturally arise in practice, become hard to formalize in this setting; and important questions which are easily answerable in informal proofs, such as whether the scope of an assumption is properly included within the scope of another assumption, whether the scopes of two assumptions are disjoint, etc., become much harder to formulate and answer.

Another thing that gets lost in sequent systems concomitantly with assumption scope is *intention*. In an informal proof, the author guides the reader through every piece of hypothetical reasoning. When the author sets out to establish a conditional

$P \Rightarrow Q$, he starts out by saying something to the effect of "To derive $P \Rightarrow Q$, assume P. Then ...". Or in a proof by contradiction, the proof might read "We will establish $\neg P$ by contradiction. To that end, suppose that P holds. Then ...". In this manner it becomes evident what the author intends to prove by a certain piece of hypothetical reasoning. The upshot is that *the explicit postulation of an assumption conveys valuable cognitive information about where the proof is heading.* Thus we see that sequent systems make a whole lot of unnecessary information tediously explicit; yet the little that *should* be made explicit, namely the postulation of assumptions in hypothetical reasoning, remains implicit, obscuring the proof's structure.[14]

7.2.3 Proof trees

The third formalization of deduction that we will discuss, popular in proof theory circles, represents proofs as trees. Specifically, a deduction of P from Γ is represented as a tree with the conclusion P at the root. At the leaves we find the various elements of Γ, possibly along with other auxiliary assumptions that are eventually discharged (more on this shortly). Each internal node contains a proposition that is obtained by the application of some inference rule to the node's children. An example is shown in Figure 7.1. Typical systems of this kind can be found in Prawitz's "Natural Deduction" [59], Van Dalen's "Logic and structure" [22], and Troelstra's "Basic Proof Theory" [68]. Some authors prefer to depict such trees with the root at the bottom, and use horizontal lines to separate conclusions from their premises, e.g. writing $\dfrac{A \quad B}{C}$ instead of

$$\begin{array}{c} C \\ \diagup \;\; \diagdown \\ A \qquad B \end{array} .$$

In order to better understand this model, it is useful to keep in mind what a tree really is formally. We tend to identify a tree with a picture of the kind we see in

[14]It is certainly possible to dress up a sequent system by introducing a keyword such as "assume", which would make it seem as if we can explicitly postulate hypotheses. But that will not get us far as long as we are manipulating sequents underneath it all. In fact it will be a misleading use of the term "assume". In HOL, for example, which is a sequent-based system, one can actually say at the prompt something like "ASSUME A", for any proposition A. However, the effect of that is to *establish* the sequent $(\{A\}, A)$. In other words, the command "ASSUME A" *proves* that A implies A. That is completely different from what "assume A" achieves in informal proofs. When we say "assume A" in an informal proof we are not *proving* anything—not yet. We are simply postulating a hypothesis. More precisely, we are tacitly manipulating the current assumption base by inserting A into it. The act of introducing A as a working hypothesis is completely different from the act of proving that A implies A; to confuse the two is almost a category mistake.

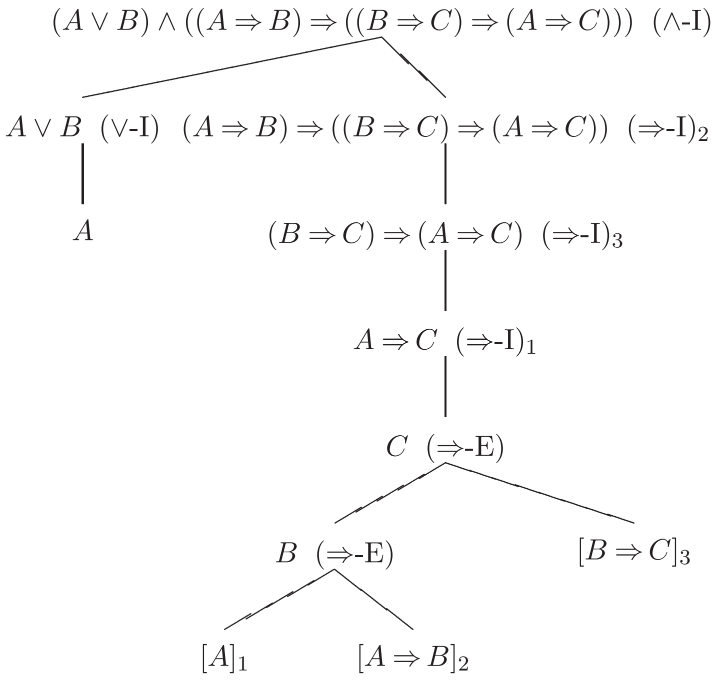


Figure 7.1: The derivation of $(A \vee B) \wedge ((A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C)))$ from A as a proof tree.

Figure 7.1, but pictures are informal objects (remember that we are discussing *formal* analyses of deduction here). Extensionally, a tree is a pointed partially ordered set (i.e., it has a "bottom", namely the root of the tree), such that the predecessors of any given node are well-ordered. In the case of proof trees, the nodes are pairs consisting of propositions and inference rules, as well as additional information to keep track of disharged hypotheses.

Thus one problem with this analysis is that it only imposes a partial order on inference rule applications and on postulated hypotheses. This is not an accurate model of informal proofs, because in an informal proof a total ordering obtains on all inferences and all hypothesis introductions (of any two of those, exactly one precedes the other). Not so with proof trees: two different topological sortings of the same proof tree will produce two distinct proofs, in the sense that we normally understand proofs; hence the loss of accuracy—different informal proofs get mapped to the same formal object. Note that it will not do to simply enforce an arbitrary ordering convention, e.g., to stipulate that children are orderered from left to right, as this will break down at the leaves. In actual proofs assumptions can be introduced at any point in the course of the argument; we do not list all of them in some fixed order in the beginning and then proceed with strict inferences.

This is not to say that a partial ordering does not give a *useful view* of a proof. By relaxing the linear ordering constraint we get a informative picture of the essential dependencies of a proof, which might prove particularly useful, for example, if we wish to parallelize a proof (whereby two inferences with no mutual dependencies may be performed concurrently). But as a formal *model* of proofs it is not intensionally accurate. The problem is particularly acute with respect to the discharged assumptions which arise in hypothetical reasoning: they are *all* maximal elements, i.e., leaves. (Actually, as we explain below, not *all* discharged hypotheses occur at the leaves. Some need to stay invisible.) That fails to reflect the order in which hypotheses are introduced in a proof; and also obfuscates the scope of a hypothesis, especially in relation to the scope of other hypotheses.

Indeed, the labelling relationship between the discharge of an assumption A at an internal node and the occurence(s) of A at the leaves has led Girard et al. [31] to claim that proof trees are "more of a graphical illusion rather than a mathematical reality" because "we have to link the crossed A with the line of the $\Rightarrow$–I rule", so that "it is no longer a genuine tree we are considering!". That remark is actually misleading, in that it confuses the leaf-to-internal-node "linking" with the partial order that is determined by rule applications (in a rigorous definition we can completely extensionalize the linking set-theoretically while preserving the tree structure); but it does attest to the clumsiness that surrounds assumption discharge in this model.

Another weakness of proof trees is that they can only depict *strict* inferences. Assumptions which do not get used cannot be handled in a uniform manner: they cannot appear at the leaves, which is where assumptions are supposed to appear in this model, for if an assumption is not used then it cannot be linked to any internal node, and hence we will no longer have a tree! Consider, for example, the derivation of $A \Rightarrow (B \Rightarrow A)$, in which the hypothesis B is never "consumed". If we were to list all assumptions at the leaves, which would be the uniform thing to do, we would get

$$\begin{array}{ccc}
A\,(1) & & B\,(2) \\
\Big| & & \\
B \Rightarrow A & \Rightarrow\text{–I},\ (2) & \\
\Big| & & \\
A \Rightarrow (B \Rightarrow A) & \Rightarrow\text{–I},\ (1) &
\end{array}$$

which is a partial order but *not* a tree. Accordingly, if the tree model is to accomodate such situations, non-strict assumptions must remain invisible. They will not appear anywhere by themselves. Instead, they will suddenly be discharged at some internal node. Thus, for example, when depicted as a tree the proof of $A \Rightarrow (B \Rightarrow A)$ must look like this:

$$
\begin{array}{c}
A\,(1) \\
| \\
B \Rightarrow A \quad \Rightarrow\text{–I, } (2) \\
| \\
A \Rightarrow (B \Rightarrow A) \ \Rightarrow\text{–I, } (1)
\end{array}
$$

Of course it is at best peculiar that we should discharge an assumption that we never introduced. What assumption does (2) label? Worse yet, the scope of such non-strict assumptions is completely obscured, for there is no starting (introduction) point, only an ending (discharge) point. Hence no path between introduction and discharge can be traced. In fact this type of situation makes it necessary to add awkward qualifications to the label rules for tagging assumptions and internal nodes (e.g., see the quoted remark below from "Basic Proof Theory" about labelling "invisible" classes of assumptions).

Another serious drawback of proof trees is the redundancy stemming from the fact that every time an assumption is used a different copy of it must appear as a tree leaf. This is not only a bad model of informal proofs (where only one "copy" of any assumption is active at any one time, and may be used multiple times), but is also quite inefficient. Any mechanized system that is based on a proof-tree model is bound to run into this problem (see the relevant discussion of LF in Section 2.2.2), and this will have an adverse impact in applications (such as PCC [52]) where proof size must be kept to a minimum. In fact it is easy to show that there are infinitely many proofs in $\mathcal{CND}$ whose tree versions are larger by an arbitrary factor n (e.g., say you pick $n = 10^{80}$; then there are infinitely many proofs which, if expressed as trees, their sizes are 10^{80} times larger than what they would be in $\mathcal{CND}$). On the other hand, it is equally easy to see that every $\mathcal{CND}$ proof is smaller than any corresponding proof-tree proof (under any sensible translation between $\mathcal{CND}$ and proof trees).

The most severe defect of this model, however, has to do with assumption discharge. For one thing, as we discussed earlier, this does not need to be made explicit, and in fact it is always tacit in informal proofs. People do not explicitly mark assumptions as "cancelled". But what is more troubling is the mechanism for effecting the discharge, namely, the tagging of leaves (assumptions) and internal nodes with matching labels. The meager extent to which this is workable hinges on visual factors and is hampered by physical and psychological limitations: if a proof is not trivial and cannot fit in one page then tracking the various matching subscripts between the leaves and the various internal nodes becomes impossible—the model does not scale. We quote from the *Handbook of Proof Theory* [12] (by a "natural deduction proof" the author means what we call a proof tree; see our earlier comments about unwarranted conflations of formal and informal notions):

> A fully constructed natural deduction proof can be very confusing to read; in particular, because of the non-local nature of natural deduction proofs, it is difficult to quickly ascertain which formulas depends [sic] on which hypotheses.

The problem is that the labelling mechanism is very low-level (reminiscent of the use of labels in programming languages, and subject to many of the same criticisms), and greatly complicates the overall system. In fact the actual rules for assumption discharge are invariably convoluted and counter-intutive. Does one discharge one particular occurence of an assumption, all occurences of the same assumption, some of them? Does one discharge only assumptions that are in the same subtree as the internal node at which the discharge occurs? The rules are usually expressed pictorially, which leaves ample room for misinterpretation. For instance, Van Dalen [22] states the rule for assumption discharge (i.e., ⇒–introduction) as follows:

$$\frac{\begin{array}{c}\not{A}\\ \vdots \\ B\end{array}}{A \Rightarrow B}\ [\Rightarrow\text{-I}]$$

Keep in mind that this is informal ("striking out" a proposition by writing a slash over it has no formal meaning; nor do the elipses $\vdots$); it is supposed to be clear and intuitive. A *rigorous* presentation of the rules would be much more difficult to follow (see, for instance, Prawitz's "Natural Deduction" [59], Section 1.2, part B). The clumsiness of the method is reflected in the presentations. Van Dalen devotes a number of pages to discussing the subtleties of assumption discharge. Even advanced texts such as Troelstra's make several qualifications to get assumption discharge straight. Here is a typical passage from Troelstra, addressing the issue of unused discharged hypotheses that we dicussed above:

> It should be noted that in the rule $\rightarrow I$ the "degenerate case", where $[A]^u$ is empty, is permitted; thus for example the following is a correct deduction:
>
> $$\cfrac{\cfrac{A^u}{B \Rightarrow A}\ v}{A \Rightarrow (B \Rightarrow A)}\ u$$

> At the first inference an empty class of occurences is discharged; we have assigned this "invisible class" a label v, for reasons of uniformity of treatment, but obviously the choice of label is unimportant as long as it differs from all other labels in use; in practice the label at the inference may be omitted in such cases.

Later in the same page:

> In applying the rule $\to I$ (assumption discharge), we do not assume that $[A]$ consists of *all* open assumptions of the form A occuring above the inference. Consider for example the following two distinct (inefficient) deductions of $A \Rightarrow (A \Rightarrow A)$:
>
> $$\dfrac{\dfrac{\dfrac{\dfrac{A^u}{A \Rightarrow A}\,v \qquad A^w}{A}}{A \Rightarrow A}\,u}{A \Rightarrow (A \Rightarrow A)}\,w \qquad\qquad \dfrac{\dfrac{\dfrac{\dfrac{A^u}{A \Rightarrow A}\,u \qquad A^v}{A}}{A \Rightarrow A}\,v}{A \Rightarrow (A \Rightarrow A)}\,w$$
>
> The formula tree in these deductions is the same, but the pattern of closing assumptions differs. In the second deduction *all* assumptions of the given form which are still open before application of an inference $\to I$ are closed simultaneously. $\cdots$

This does not strike me as "natural deduction", in any sense of "natural". There is little connection to actual proofs.

We will now consider two problems of proof trees that carry over to their λ-calculus analogues through the Curry-Howard correspondence. (The reader is expected to have a basic understanding of the Curry-Howard isomorphism in order to understand the following discussion; the classic exposition is Howard's article "The formulae-as-types notion of construction".[15]) These are problems which do not get remedied by the binding power of the λ; they run deeper, to the costructive interpretation of the logical connectives. They show that, as they stand, proof trees and their Curry-Howard counterparts are not adequate formalizations of the informal notion of proof: in one direction, they are *not fine enough*, meaning that there are informal proofs which cannot be faithfully represented as proof trees (or λ-terms); in the other direction, they

[15] Other references include the books by Girard et al. [31], by Goubault-Larrecq and Mackie [34], and by Troelstra and Schwichtenberg [68].

are *too fine*, meaning that they formally distinguish proofs which informally would be deemed indistinguishable.

In one direction, consider the derivation of $A \wedge A$ from $A \wedge B$. Informally, this would proceed as follows:

> From $A \wedge B$, we may infer A (using $\wedge$-elimination). From A, we infer $A \wedge A$ (using $\wedge$-introduction). (7.2)

Note that, naturally, we *only derive A once*. But, as a proof tree, the cleanest way to represent this deduction is as follows:

$$\dfrac{\dfrac{A \wedge B}{A} \qquad \dfrac{A \wedge B}{A}}{A \wedge A}$$

The glaring redundancy here, of course, is the duplicate derivation of A from $A \wedge B$. But the problem is really philosophical, going beyond efficiency: there is no formal counterpart (proof tree) that is a faithful representation of the informal proof (7.2). For a faithful representation would entail that any meaningful statement we can make about the informal object i would be preserved *salva veritate* (without change of truth value) once appropriately translated so as to become a statement about the formal counterpart of i. That is clearly not the case here if we regard the foregoing proof tree as the formal counterpart of (7.2). The statement "A is derived from $A \wedge B$ once" is true for (7.2), but false for the proof tree.

Moreover, the proof tree above cannot be simplified in any way so as to avoid the duplicate derivation of A. It is, in proof-theoretic terms, in *normal form*: it is as simple as it can possibly get. Proof trees in normal form are advertised as free of redundancies and extraneous steps. That is clearly not true in view of this example. Normal form for proof trees does *not* capture the intuitive notion of redundancy-free proofs. Under the Curry-Howard correspondence, the analogue of this proof tree in the λ-calculus is the term

$$\mathbf{pair}(\mathbf{first}(x_{A\times B}), \mathbf{first}(x_{A\times B})) \tag{7.3}$$

or perhaps

$$\lambda\, x_{A\times B} \,.\, \mathbf{pair}(\mathbf{first}(x_{A\times B}), \mathbf{first}(x_{A\times B}))$$

if we wish to close over ("discharge") $A \wedge B$ (proving the theorem $A \wedge B \Rightarrow A \wedge A$, rather than deriving $A \wedge A$ from $A \wedge B$). A λ-calculus term, of course, is in normal form iff it contains no β-redex, i.e., no subterm of the form $\mathbf{app}(\lambda\, x_\tau \,.\, M, N)$, and no pairs surrounded by a projection, i.e., no subterms of the form $\mathbf{first}(\mathbf{pair}(M, N))$ or $\mathbf{second}(\mathbf{pair}(M, N))$. By the Curry-Howard isomorphism, a term M is in such a normal form iff the corresponding proof tree is in (proof-theoretic) normal form. Thus we can see that the proof tree above is really irreducible by noting that the corresponding λ-term (7.3) contains no redexes.

The second problem lies in the fact that in the typed λ-calculus, *variables are identified not with assumptions but with proofs of assumptions*. This is a very important point that gets to the heart of a fundamental difference between the Curry-Howard approach and DPLs. Several authors describing the Curry-Howard correspondence state that variables correspond to assumptions (e.g., Girard et al., [31]). That is incorrect, as an assumption and a proof of an assumption are two very different things; and under the Curry-Howard correspondence variables correspond to proofs of assumptions, not to assumptions. This can be easily proven simply by noticing that for *each* proposition (type) τ there are infinitely many *distinct* variables $x_\tau, y_\tau, z_\tau, x'_\tau, \ldots$. More elaborately:

(1) Under the Curry-Howard-isomorphism, (different) terms correspond to (different) proofs (perhaps of the same proposition, but different proofs nevertheless).

(2) Pick any proposition (type) τ. The variables x_τ and y_τ are different terms, hence they represent different proofs (of the same proposition, τ).

(3) Now if variables were identified with assumptions, i.e., with propositions, then x_τ and y_τ would be indistinguishable, contradicting (2) above.

In fact the identification of typed variables with proofs of propositions rather than with mere propositions is crucial for Heyting's constructive interpretation of the logical connectives: remember that a proof of a conditional $A \Rightarrow B$ is a computable function $\lambda\, x_A \,.\, M$ that maps an arbitrary *proof* of A (precisely the variable x_A) into a proof of B (the term M, which must thus be of type B).

Now what does that entail about proof trees (and through the Curry-Howard-correspondence, the terms of the typed λ-calculus) as formalizations of the notion of proof? It has negative ramifications: we are in a position where we must distinguish between formal objects which would not be informally distinguished. For instance, consider the terms (proofs)

$$\lambda\, x_A \,.\, \lambda\, y_A \,.\, x_A \tag{7.4}$$

and

$$\lambda\, x_A \,.\, \lambda\, y_A \,.\, y_A. \tag{7.5}$$

These are distinct terms (they are not α-convertible), representing two *distinct* proofs of the same proposition, namely $A \Rightarrow (A \Rightarrow A)$. Heyting's constructive interpretation of conditionals helps to understand why these are two different proofs of $A \Rightarrow (A \Rightarrow A)$: in (7.4), we are given a proof x_A of A, then we are given another proof y_A of it, and we finally decide to *use the first proof*, which might well have significant computational differences from the second one (e.g., one might be much more efficient than the other). By contrast, in (7.5) we are given a proof of A, then another proof of it, and finally we decide to *use the second proof*, disregarding the first. But in mathematical practice these two proofs would (and should) be conflated, because one *postulates propositions, not proofs of propositions*. The customary mode of hypothetical reasoning is

$$\text{“Assume that } A \text{ holds. Then } \cdots A \cdots \text{.”}$$

not

$$\text{“Assume we have a proof } \Pi \text{ of } A\text{. Then } \cdots \Pi \cdots A \cdots \text{.”}$$

Clearly, to assume that something is true is not the same as to assume that we have a proof of it.

In a DPL this problem would not arise because both (7.4) and (7.5) (as well as the infinitely many other terms which can be gotten by using different variables in place of x and y) would be conflated to

$$\textbf{assume } A \,.\, \textbf{assume } A \,.\, A$$

which captures perfectly well how this proposition would be proved in actuality.

Incidentally, this shows that there cannot be an isomorphism between a customary typed λ-calculus and a DPL, for we cannot have a bijection: infinitely many λ-terms would be mapped to a single DPL deduction. It is possible to obtain a bijection by collecting variables of the same type into equivalence classes, taking the compatible closure of this equivalence relation, and then mapping equivalence classes of the resulting relation to DPL deductions. But then the relationship of β-reduction on one hand and proof normalization on the other becomes complicated, since we now have to consider β-reduction between equivalence classes of λ-terms if we wish to preserve the Curry-Howard isomorphism in any meaningful sense. What is immediately apparent is that even in the absence of proof composition, DPLs are not just notational variants of typed λ-calculi. Unlike proof trees, there is no bijective correspondence between DPL deductions and terms of the typed λ-calculus. And from the foregoing discussion, that is a good thing—it is closer to informal mathematical practice.

7.2.4 Quantifier reasoning

Our discussion so far has been restricted to the propositional fragments of the reviewed systems. When we come to quantifier reasoning we are presented with an additional set of difficulties. I will single out two classes of problems below. Since all three formalizations (FRH, sequent, and proof-tree systems) are suspect to these problems, the discussion will be generic and should be understood to apply to all three analyses.

It will be seen that the source of these problems is the view of a deduction as a static object with a fixed meaning. Once we separate syntax from meaning we are free to equip deductions with *evaluation semantics* that averts these problems.

The first problem has to do with tedious free-variable restrictions on the applicability of various quantifier rules. An example is universal specialization, i.e., the derivation of $P[t/x]$ from $(\forall x)P$. Most systems require t to be "safe" for x in P in order to avoid variable capture. Likewise for existential generalization, the inference of $(\exists x)P$ from $P[t/x]$. Clearly, this is a trivial matter and should be of no concern to the person expressing the proof, just as it is of no concern to a mathematician presenting a proof in a journal (did you ever read a mathematical article where the author explicitly took care to satisfy free-variable conditions when specializing a universal generalization?) Propositions are understood modulo alphabetic equivalence, and this should be reflected in a formal system. This is not a deep point, yet most formalizations of deduction do not observe it (Howard-Curry *representations* of such formalizations handle this problem easily, but they present other problems).

The second class of problems arises in connection with the two remaining kinds of quantifier reasoning: universal generalization ($\forall$–introduction), and existential specialization ($\exists$–elimination). We will discuss each in turn. Asides from specialized techniques such as various forms of induction, the most common method for establishing a universal generalization $(\forall x)P$ is to consider an *arbitrary* individual a and show that P holds for it. The idea of course is that because no special assumptions about the nature of a have been made, a may well be *any* element (in the underlying universe of discourse) whatsoever. Thus showing $P(a)$ amounts to showing that P holds for *any* element, which is to say $(\forall x)P$. Now the interesting aspect of such informal proofs that is not captured by any of the formalizations we have considered is that the parameter a has *scope*, namely the *subdeduction* of $P(a)$. Any use of a that might had been in effect at the point when the author says "Consider an arbitrary $a \cdots D \cdots$", where D is the deduction that derives $P(a)$, becomes masked by this new use of a. The masking remains in effect throughout D. Following D, the old use of a, if one existed, is restored.

This is of course similar to lexical scoping in programming languages, but there are important differences. In a programming language masking amounts to rebinding a variable to some new value; e.g., when we say `(let (`x E_1`)` E_2`)`, we *bind* x to the value of E_1 and proceed to evaluate E_2. But when we say "pick an arbitrary a and consider $\cdots$", what does a come to denote? Certainly not any particular element in the universe of discourse. Thus we see that it is only the syntactic notion of scope that is similar in both cases; the evaluation semantics must be different.

The important point is that deductions of the form

"pick an arbitrary $a \cdots$" (U)

introduce scope for a, and the formal analysis should capture and support this. That is not the case with any of the three formalizations we have examined. None of them provide any notion of scope for deductions of the form (U). What one must do instead is prove $P(x)$ individually, *for some x that does not occur free in the assumptions*, and then use the rule of universal generalization to conclude $(\forall x)P$. This caveat about free occurences of x in the assumptions is unnecessary and does not reflect the way in which such proofs are carried out in practice. If scope is taken into account properly then we can do away with the caveat.

Similar considerations apply to existential specialization. Suppose we have established a statement of the form $(\exists x)P$. One can use such a statement to derive further results by reasoning as follows: "We know that P holds for *some* element. Let a be the name of that element. Then $\cdots\ D\ \cdots$", where D is a deduction that infers some statement A, which is the final upshot. This is a very common form of reasoning, and it, too, is characterized by scope: a has a clearly delineated range, namely, the *subdeduction D*. Any prior use of a becomes obsolete at the beginning of this subdeduction, and is restored at the end. That is the proper way to formalize this form of reasoning. Instead, all of the analyses that we have considered impose an array of cumbersome restrictions on free occurences of a in assumptions, in the conclusion A, and elsewhere. Once again, this does not reflect common mathematical reasoning.

Our formalization of natural deduction as a DPL has the closest kinship to a family of systems that date back to work done by members of "the Polish school" in the late 1920s. A brief history of this line of work is given in Appendix C of Prawitz's "Natural Deduction". It was Łukasiewics who first expressed dissatisfaction with the FRH approach, as far back as 1926, pointing out that it does not capture informal mathematical reasoning. Jáskowski in 1929 presented the first deduction system that attempted to formalize common mathematical inference. It was the first in a long line of systems that we will refer to as "box systems". Deductions in such systems are visually characterized by the presence of nested boxes. Every introduction of an assumption A starts off another box, which delineates the scope of A. Within the box of A, one proceeds to derive other propositions which may depend on A, as well as on other assumptions which were introduced earlier, and possibly introduce additional assumptions (which will open other boxes, nested inside the box of A). Finally, immediately below the box of A we write the proposition obtained by discharging A.

Box systems have been very successfull pedagogically; most logic textbooks have traditionally used some such system to teach deduction. Some well-known variants that have been widely used can be found in books by Fitch [25], Kalish and Montague [41], Copi [17], Quine [60], Suppes [67], Lemmon [46], or, more recently, Bergman et al. [5].

The common trait of all these systems is that they pay proper attention to hypothetical reasoning and assumption scope; their pedagogical success is a testament to the importance of those concepts. But when it comes to reasoning with quantifiers box systems are subject to the same criticisms as the other systems (unnecessary eigenvariable restrictions, etc.)

However, insofar our concern is a formal analysis of the concept of deduction, the most serious problem is that deductions in box systems are not formal objects. They are based on pictorial devices, and as a result, all of the important ideas remain intuitively clear but informal. Attempts to extensionalize such systems have been heavily syntactic, and thus clumsy and subject to many of the foregoing criticisms. Essentially, in doing away with boxes they have all relied on some type of labelling or tagging mechanism to keep track of assumption dependencies. Jáskowski [50], for example, prefixes every occurence of a proposition P with a list of numerals that indicate the assumptions on which that occurence of P depends. A deduction then becomes a finite sequence of such decorated propositions, where each member of the sequence is either taken for granted or obtained from previous members by some inference rule. The inference rules are designed to keep track of and manipulate the numeric prefixes appropriately, and are thus unduly cumbersome. Quine uses a "flagging" mechanism that is similar in spirit but even more convoluted. In conclusion we might say that box systems have the right idea, at least concerning propositional reasoning, but have been difficult to formalize while retaining their intuitive advantages.

"Though this be madness, yet there is method in 't."
William Shakespeare, Hamlet

8 The $\lambda\mu$-calculus

In this chapter we introduce the $\lambda\mu$-calculus, which can be viewed as a uniform underlying framework for DPLs. In Section 8.1 and Section 8.2 we define its syntax and evaluation semantics. In Section 8.3 we call attention to some of its novel characteristics and the rationale behind them. The next two sections develop the rudiments of the theory and metatheory of the $\lambda\mu$-calculus. Section 8.6 introduces some notational conventions and syntax sugar that we will use in the sequel. Section 8.7 discusses the subject of interpreting the $\lambda\mu$-calculus, and presents three such interpreters. Finally, Section 8.8 presents an introductory example of a $\lambda\mu$ system, and introduces the notion of credible computation. Several additional examples will be given in the next chapter.

8.1 Syntax

There are three syntactic categories in the $\lambda\mu$-calculus: *deductions*, *expressions*, and *phrases*. The first two are the important ones; a phrase is simply either a deduction or an expression. Put differently, $Phr = Exp \cup Ded$, where we write Phr, Exp, and Ded for the sets of all phrases, expressions, and deductions, respectively. We use the letters D and E to range over deductions and expressions, respectively; M and N will range over phrases. The letter I ranges over the set Ide of *identifiers*; we assume that Ide is countably infinite. The letter c ranges over an unspecified set C of *constants*; the only assumption we will make about C is that it is disjoint from Ide and that it contains the constant claim (as a convention, we will use sans serif font for constants).

We will use the letter Ξ as a metavariable ranging over the variables D, E, and M (thus Ξ has only three possible values.) We write *kwd* for an unspecified keyword ("token", or "terminal" in parsing terminology); keywords will be written in boldface. Finally, the symbol $\overrightarrow{I}$ ($\overrightarrow{E}$, etc.) ranges over finite sequences—possibly empty—of identifiers (expressions, etc.). The following grammar specifies the abstract syntax of the $\lambda\mu$-calculus:

$$
\begin{array}{rl}
D \in Ded & \text{Deductions} \\
E \in Exp & \text{Expressions} \\
M, N \in Phr & \text{Phrases}
\end{array}
$$

$$D \;::=\; \mathbf{dapp}(E, \overrightarrow{M}) \;\{\; \mid kwd_1(\overrightarrow{\Xi}_1) \mid \cdots \mid kwd_n(\overrightarrow{\Xi}_n)\} \tag{8.1}$$

$$E \;::=\; c \mid I \mid \mu\, \overrightarrow{I}\,.\, D \mid \lambda\, \overrightarrow{I}\,.\, E \mid \mathbf{app}(E, \overrightarrow{M}) \tag{8.2}$$

$$M \;::=\; E \mid D \tag{8.3}$$

The braces $\{\cdots\}$ in 8.1 indicate that the enclosed productions are optional. Thus equation 8.1 is really a "meta-clause", in that it allows for any finite number $n \geq 0$ of *special deductive forms* $kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n)$. An example of a special deductive form is $\mathbf{assume}(M, D)$. This form matches the schema $kwd(\Xi_1, \Xi_2)$, where *kwd* is **assume**, Ξ_1 is M, and Ξ_2 is D. Once we instantiate the various kwd_i and Ξ_i as desired, equations 8.1—8.3 single out a fixed, well-defined set of abstract syntax trees. When there are no special deductive forms we speak of a *pure $\lambda\mu$ system*; otherwise we have an *augmented* $\lambda\mu$ system. Thus in a pure $\lambda\mu$ system every deduction is of the form $\mathbf{dapp}(E, \overrightarrow{M})$. We will see that pure systems are sufficient for most purposes; special deductive forms are never strictly necessary. We can get sound and complete systems for many important logics, including classical first-order logic, in the setting of the pure $\lambda\mu$-calculus (see Section 9.5). However, some special deductive forms are occasionally essential for directly capturing the structure of certain modes of reasoning, especially in natural deduction. Nevertheless, even in those cases we have observed that the number of special deductive forms can be kept to a minimum; once a fundamental form or two have been chosen, other forms become expressible as syntax sugar. In this entire document we will only consider two special deductive forms.

A particular *$\lambda\mu$ system* is obtained when we fix a set of constants C, a set of *δ-rules* Δ specifying the behavior of the constants (to be discussed shortly), and a number $n \geq 0$ of special deductive forms $kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n)$. Accordingly, we will write a $\lambda\mu$ system $\mathcal{L}$ as

$$\mathcal{L} = (kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n); C; \Delta) \tag{8.4}$$

or simply $\mathcal{L} = (kwd_1, \ldots, kwd_n; C; \Delta)$ when there is no risk of confusion. As we remarked above, when there are no special deductive forms we say that $\mathcal{L}$ is *pure*, and we write $\mathcal{L} = (C; \Delta)$. A pure system is completely determined by its constants and its δ-rules.

Note that the regular λ-calculus

$$E \;::=\; c \mid I \mid \lambda\, \overrightarrow{I} \,.\, E \mid \mathbf{app}(E, \overrightarrow{E}) \tag{8.5}$$

is directly embedded into the $\lambda\mu$-calculus. That is, every expression generated by 8.5 is also an expression of the $\lambda\mu$-calculus. We will see that the regular reduction semantics of the λ-calculus are also wholly preserved.

Expressions of the form $\mu\, I_1, \ldots, I_k \,.\, D$ and $\lambda\, I'_1, \ldots, I'_n \,.\, E$ are called *methods* and *functions*, respectively. These are abstractions of deductions and computations, respectively. We refer to $I_1, \ldots, I_k$ and $I'_1, \ldots, I'_n$ as the *parameter* lists of the method and the function. We require that parameter lists have no duplicate entries. Expressions and deductions of the form $\mathbf{app}(E, \overrightarrow{M})$ and $\mathbf{dapp}(E, \overrightarrow{M})$ are called function and method *applications*, respectively.

Free and bound occurences of identifiers are defined as usual. Specifically, the set of identifiers that have free occurences in a phrase M, denoted $FV(M)$, is defined as follows:

$$\begin{array}{rcl}
FV(\mathbf{dapp}(E, \overrightarrow{M})) & = & FV(\mathbf{app}(E, \overrightarrow{M})) \;=\; FV(E) \cup FV(\overrightarrow{M}) \\
FV(kwd(\Xi_1, \ldots, \Xi_n)) & = & FV(\Xi_1) \cup \cdots \cup FV(\Xi_n) \\
FV(c) & = & \emptyset \\
FV(I) & = & \{I\} \\
FV(\mu\, I_1, \ldots, I_k \,.\, D) & = & FV(D) - \{I_1, \ldots, I_k\} \\
FV(\lambda\, I_1, \ldots, I_k \,.\, E) & = & FV(E) - \{I_1, \ldots, I_k\}
\end{array}$$

where $FV(\overrightarrow{M})$ denotes $FV(M_1) \cup \cdots \cup FV(M_k)$ whenever $\overrightarrow{M} = M_1, \ldots, M_k$. The set of identifiers that have *bound* occurences in a phrase F, denoted $BV(F)$, is defined thus:

$$\begin{array}{rcl}
BV(\mathbf{dapp}(E, \overrightarrow{M})) & = & BV(\mathbf{app}(E, \overrightarrow{M})) \;=\; BV(E) \cup BV(\overrightarrow{M}) \\
BV(kwd(\Xi_1, \ldots, \Xi_n)) & = & BV(\Xi_1) \cup \cdots \cup BV(\Xi_n) \\
BV(c) & = & \emptyset \\
BV(I) & = & \emptyset \\
BV(\mu\, I_1, \ldots, I_k \,.\, D) & = & \{I_1, \ldots, I_k\} \cup BV(D) \\
BV(\lambda\, I_1, \ldots, I_k \,.\, E) & = & \{I_1, \ldots, I_k\} \cup BV(E)
\end{array}$$

Phrases which differ only in the names of their bound variables are called *alphabetic variants*, and will be identified. That is, we will consider two phrases to be identical iff they are alphabetically convertible. When $\overrightarrow{N} = N_1, \ldots, N_{k_1}$, $\overrightarrow{I} = I_1, \ldots, I_{k_2}$, $k_1 = k_2 \geq 0$, and the identifiers $I_1, \ldots, I_{k_2}$ are distinct, we define $M[\overrightarrow{N}/\overrightarrow{I}]$ as the phrase obtained from M by simultaneously replacing every free occurence of I_j by N_j, $j = 1, \ldots, k_1 = k_2$. In general this replacement can result in variable capture, but since alphabetic variants are considered identical, we can always rename the bound identifiers of M so as to make this impossible. If $k_1 \neq k_2$ or the identifiers $I_1, \ldots, I_{k_2}$ are not distinct then $M[\overrightarrow{N}/\overrightarrow{I}]$ is undefined.

8.2 Semantics

For the remainder of this section fix a $\lambda\mu$ system $\mathcal{L}$ of the form 8.4. We will assume that the set of constants C is partitioned into three parts, the set $Meth_C$ of *primitive methods*, the set Fun_C of *primitive functions*, and the set Val_C of *primitive values*. Furthermore, a subset $Sent_C \subseteq Val_C$ must be singled out as the set of *sentences*. Intuitively, the sentences will be the "statements" or "assertions" that one can formulate in the system. The letters ϕ, ψ, V and S will range over primitive functions, primitive methods, primitive values, and sentences, respectively. An *assumption base* (or *context*) β will simply be a finite set of sentences, so that

$$\beta \subseteq Sent_C \subseteq Val_C \subseteq C.$$

Finally, we will require that every primitive method ψ have a unique non-negative integer $r(\psi) \geq 0$ associated with it and known as its *arity*. Likewise, every primitive function ϕ must have a given positive arity $r(\phi) > 0$.

The semantics of $\mathcal{L}$ are given via rules that establish judgments of the form

$$\beta \vdash_{\mathcal{L}} M \rightsquigarrow N$$

which may be read as "In the the assumption base β, M evaluates to N", or "N is *derivable* from M relative to β", or "β proves that we can derive N from M". When $\mathcal{L}$ is understood or immaterial we simply write $\beta \vdash M \rightsquigarrow N$. We write $\beta \vdash_{\mathcal{L}} M \rightsquigarrow^* N$ to mean that $\beta \vdash_{\mathcal{L}} M \rightsquigarrow N$ or $M = N$.

The rules are divided into three groups: the core rules, which specify the semantics of the standard $\lambda\mu$ constructs; the δ-rules, which specify the behavior of the constants; and the special-form rules, which specify the semantics of the special deductive forms of $\mathcal{L}$. Of course in a pure $\lambda\mu$ system there are no special deductive forms and the last group of rules is empty.

$$\frac{}{\beta \vdash \mathbf{app}(\lambda\, \overrightarrow{I}\,.\,E, \overrightarrow{M}) \rightsquigarrow E[\overrightarrow{M}/\overrightarrow{I}\,]}\;[\text{R1}] \qquad \frac{}{\beta \vdash \mathbf{dapp}(\mu\, \overrightarrow{I}\,.\,D, \overrightarrow{M}) \rightsquigarrow D[\overrightarrow{M}/\overrightarrow{I}\,]}\;[\text{R2}]$$

$$\frac{\beta \vdash E \rightsquigarrow E'}{\beta \vdash \mathbf{app}(E, \overrightarrow{M}) \rightsquigarrow \mathbf{app}(E', \overrightarrow{M})}\;[\text{R3}] \qquad \frac{\beta \vdash E \rightsquigarrow E'}{\beta \vdash \mathbf{dapp}(E, \overrightarrow{M}) \rightsquigarrow \mathbf{dapp}(E', \overrightarrow{M})}\;[\text{R4}]$$

$$\frac{\beta \vdash M_i \rightsquigarrow M_i'}{\beta \vdash \mathbf{app}(E, M_1, \ldots, M_i, \ldots, M_k) \rightsquigarrow \mathbf{app}(E, M_1, \ldots, M_i', \ldots, M_k)}\;[\text{R5}]$$

$$\frac{\beta \vdash E_i \rightsquigarrow E_i'}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, E_i, \ldots, M_k) \rightsquigarrow \mathbf{dapp}(E, M_1, \ldots, E_i', \ldots, M_k)}\;[\text{R6}]$$

$$\frac{\beta \vdash D_i \rightsquigarrow S \qquad \beta \cup \{S\} \vdash \mathbf{dapp}(E, M_1, \ldots, S, \ldots, M_k) \rightsquigarrow N}{\beta \vdash \mathbf{dapp}(E, M_1, \ldots, D_i, \ldots, M_k) \rightsquigarrow N}\;[\text{R7}]$$

$$\frac{\beta \vdash E \rightsquigarrow N}{\beta \vdash \lambda\, \overrightarrow{I}\,.\,E \rightsquigarrow \lambda\, \overrightarrow{I}\,.\,N}\;[\text{R8}] \qquad \frac{\beta \vdash D \rightsquigarrow N}{\beta \vdash \mu\, \overrightarrow{I}\,.\,D \rightsquigarrow \mu\, \overrightarrow{I}\,.\,N}\;[\text{R9}]$$

$$\frac{\beta \vdash D \rightsquigarrow S}{\beta \cup \beta' \vdash D \rightsquigarrow S}\;[\text{R10}] \qquad \frac{\beta \vdash M_1 \rightsquigarrow M_2 \qquad \beta \vdash M_2 \rightsquigarrow M_3}{\beta \vdash M_1 \rightsquigarrow M_3}\;[\text{R11}]$$

$$\frac{}{\{S\} \vdash \mathbf{dapp}(\mathsf{claim}, S) \rightsquigarrow S}\;[\text{R12}]$$

Figure 8.1: Semantics of the pure $\lambda\mu$-calculus.

8.2.1 Core semantics

The rules defining the core semantics of $\lambda\mu$ systems appear in Figure 8.1. Rules [R1] and [R2] model the customary notion of *reduction* (this is the usual notion of β-reduction, but we reserve the letter β for assumption bases). Rules [R3]—[R6], [R8], and [R9] are simple compatibility rules. Rule [R10] is a dilution rule. Rule [R11] ensures that the derivability relation (with respect to a fixed assumption base β) is transitive. Finally, rule [R12] fixes the semantics of the primitive method claim.

[R7] is the most distinctive rule of the $\lambda\mu$-calculus. It is the rule that allows for *inference composition*: deriving conclusions that will be used in subsequent deductions. To understand what it says, think of the expression E as an inference rule such as

Modus Ponens and read the rule backwards. The rule states that if the i^{th} argument of a method application is a deduction (D_i), then the application at hand can take the result of that deduction (S) for granted That is, the application can be evaluated in $\beta \cup \{S\}$, instead of just β. We will see many examples of this rule in action soon.

8.2.2 Semantics of constants

Recall that the set of constants C is made up of three pairwise disjoint sets, $Meth_C$, comprising the primitive methods; Fun_C, containing the primitive functions; and Val_C, the set of primitive values. The latter are uninterpreted constants that are intended as the results of computations and deductions; there are no evaluation rules for them. The only assumption we will make is that the equality relation on Val_C is decidable. In order to simplify the presentation, we will often treat the elements of Val_C as abstract objects; and we will allow for any number of them, even uncountably many. For a real calculus, of course, we should instead work with concrete, finite representations of abstract objects.

Semantics of primitive methods

For every primitive method ψ with arity n there is a given set of *δ-rules*

$$\Delta_\psi \subseteq \mathcal{P}_\infty(Sent_C) \times C^n \times Sent_C$$

where $\mathcal{P}_\infty(Sent_C)$ is the set of all finite subsets of $Sent_C$. Thus every δ-rule $R \in \Delta_\psi$ is a triple of the form

$$R = \langle \{S_1, \ldots, S_{k_R}\}, \langle c_1, \ldots, c_n \rangle, S \rangle \tag{8.6}$$

where the number k_R may depend on R. We say that $S_1, \ldots, S_{k_R}$ are the rule's *premises*; $c_1, \ldots, c_n$ are its *arguments*; and S is its *conclusion*. If

$$\{S_1, \ldots, S_{k_R}\} \subseteq \{c_1, \ldots, c_n\}$$

for *every* $R \in \Delta_\psi$ we say that ψ is *compositional*, for reasons that will be explained later. Further, if every rule has the same number of premises, i.e., if $k_{R_1} = k_{R_2}$ for all R_1, R_2 in Δ_ψ, we say that ψ is *uniform*. All of the primitive methods we will encounter in the sequel will be uniform. A primitive method all of whose rules have zero premises is called an *axiom*. Note that axioms are trivially uniform and compositional. Also note that primitive methods are finitary, in that every rule has a finite number of premises. Finally, for any δ-rule R, we write $Prem(R)$, $Arg(R)$, and $Con(R)$ to denote the set of premises, the tuple of arguments $\langle c_1, \ldots, c_n \rangle$, and the conclusion of R, respectively.

For every rule $R \in \Delta_\psi$ of the form 8.6 we stipulate that the following judgment holds:

$$\{S_1, \ldots, S_{k_R}\} \vdash \mathbf{dapp}(\psi, c_1, \ldots, c_n) \leadsto S$$

or, equivalently,

$$\mathit{Prem}(R) \vdash \mathbf{dapp}(\psi, \mathit{Arg}(R)) \leadsto \mathit{Con}(R).$$

Such a judgment will be called a *δ-evaluation axiom*. In addition, we impose the following requirements:

PM1 The conclusion of a rule is uniquely determined by its arguments. More precisely, $S = S'$ whenever $\langle \beta_1, \langle c_1, \ldots, c_n \rangle, S \rangle \in \Delta_\psi$ and $\langle \beta_2, \langle c_1, \ldots, c_n \rangle, S' \rangle \in \Delta_\psi$.

PM2 The following problem is mechanically solvable: given an arbitrary assumption base β and a list of constants $c_1, \ldots, c_n$, determine whether or not there is a δ-rule in Δ_ψ whose arguments are $c_1, \ldots, c_n$ and whose premises are contained in β; and if there is such a rule, produce its conclusion (there may be several such rules in Δ_ψ, but they will all have the same conclusion by virtue of the previous requirement).

Of course Δ_ψ will usually be infinite and we will not be able to specify it by exhaustively enumerating its members. Rather, we will usually specify it by a small number (usually one) of *rule schemas* of the form

$$\{\pi_1, \ldots, \pi_k\} \vdash \mathbf{dapp}(\psi, \pi'_1, \ldots, \pi'_n) \leadsto \pi \tag{8.7}$$

where the πs are patterns containing variables that range over C. It will then be understood that Δ_ψ contains all and only those triples $\langle \{S_1, \ldots, S_k\}, \langle c_1, \ldots, c_n \rangle, S \rangle$ such that $S_1, \ldots, S_k, c_1, \ldots, c_n$, and S can be obtained from $\pi_1, \ldots, \pi_k, \pi'_1, \ldots, \pi'_n$ and π by consistently instantiating pattern variables with specific constants. As an example, the δ-rules for Modus Ponens (mp) in simple propositional logic will be specified with the single schema $\{P \Rightarrow Q, P\} \vdash \mathbf{dapp}(\mathsf{mp}, P \Rightarrow Q, P) \leadsto Q$. Here P and Q are variables ranging over propositions, which serve as the sentences of the system. Every rule in Δ_{mp} can be obtained from the above schema by fixing values for P and Q. Note that mp is uniform and compositional. That will be the case for most of the primitive methods we will encounter.

Semantics of primitive functions

When it comes to primitive functions it is useful to allow for rules of the form

$$\beta \vdash \mathbf{app}(\phi, M_1, \ldots, M_k) \leadsto N$$

whereby a primitive function ϕ manipulates entire phrases rather than just constants (as primitive methods do). However, embracing such rules in an unrestricted fashion can easily lead to unpleasant consequences. Most notably, it can destroy confluence, even when the rules themselves determine unique functions; and it can considerably complicate evaluation.

In particular with regard to evaluation, in order to handle primitive applications in a uniform manner across different reduction strategies (by-name, by-value, etc.), we would like the following condition to obtain: when evaluating a primitive application

$$\mathbf{app}(\phi, M_1, \ldots, M_k)$$

(in some β), we should be able to go ahead and evaluate the arguments $M_1, \ldots, M_k$, obtain values $v_1, \ldots, v_k$ for them (where each "value" v_i is an expression in some kind of normal form), and *then* apply the primitive ϕ to the obtained values. Moreover, the result of the application should itself be in normal form, so that it does not require any further evaluation.

The standard notion of normal form that we will consider in this document will be that of "weak normal form", which is determined syntactically and is thus independent of any particular semantics for the $\lambda\mu$-calculus. Specifically, a phrase M is in weak normal form iff it is either a constant c, or of the form $\lambda \overrightarrow{I} \,.\, E$ or $\mu \overrightarrow{I} \,.\, D$ for *arbitrary* D and E (strictly speaking we should also include identifiers I and applications of the form $\mathbf{app}(I, \overrightarrow{M})$ with $\overrightarrow{M}$ in weak normal form, but we will only be interested in closed phrases—with no free identifiers—where such cases do not arise). Unlike the definition of "normal form" which we will formulate later on and which hinges on a specific set of reduction rules, this is a purely syntactic criterion; it is independent of evaluation semantics and assumption bases.

Our $\lambda\mu$-interpreters will be required to evaluate phrases up to weak normal form. Accordingly, in order to ensure the aforementioned condition we should at the very least require that primitive applications operate on weak normal forms and produce weak normal forms. However, this is not sufficient to guarantee unproblematic behavior. As an example, consider a primitive equality function $=$ along with the rules

$$\frac{}{\beta \vdash \mathbf{app}(=, v, v) \rightsquigarrow \mathsf{true}}$$

$$\frac{}{\beta \vdash \mathbf{app}(=, v_1, v_2) \rightsquigarrow \mathsf{false}}$$
$$\text{whenever } v_1 \neq v_2$$

where v ranges over expressions in weak normal form and true and false are primitive values. These rules successfully determine a unique, decidable function that takes an assumption base and a pair of expressions in weak normal form and produces an expression in weak normal form (either the constant true or false). However, the rules destroy confluence, as can be seen by considering the expression

$$\mathbf{app}(\lambda\, x, y\,.\,\mathbf{app}(=, \lambda\, z\,.\,x, \lambda\, z\,.\,y), \mathsf{true}, \mathsf{true}).$$

By reducing the outermost λ redex, one obtains $\mathbf{app}(=, \lambda\, z\,.\,\mathsf{true}, \lambda\, z\,.\,\mathsf{true})$, which then reduces to true by the foregoing δ-rules. However, by first reducing the application of $=$ to $\lambda\, z\,.\,x$ and $\lambda\, z\,.\,y$ we obtain

$$\mathbf{app}(\lambda\, x, y\,.\,\mathsf{false}, \mathsf{true}, \mathsf{true})$$

(since $\lambda\, z\,.\,x$ and $\lambda\, z\,.\,y$ are syntactically distinct), and with one more reduction we get false. Hence one and the same expression produces two different results.[1]

In the above example the culprit was the presence of the free variables in the arguments to the primitive $=$, so we should require the arguments to a primitive application to be *closed* expressions in weak normal form. However, we will go a step further and require that all arguments in a δ-rule of a primitive function ϕ must be constants; and we will require that the result must be a closed expression in weak normal form.

Thus we require that for every primitive function ϕ of arity $n > 0$ we are given a set Δ_ϕ of triples of the form

$$R = \langle \beta, \langle c_1, \ldots, c_n \rangle, E \rangle \tag{8.8}$$

where β is an arbitrary assumption base called the *context* of the rule; $c_1, \ldots, c_n$ are constants, called the *arguments* of R; and E is a closed expression in weak normal form called the *result* of R. We say that ϕ is *context-independent* if

$$\langle \beta', \langle c_1, \ldots, c_n \rangle, E \rangle \in \Delta_\phi$$

for *every* $\beta' \subseteq \mathit{Sent}_C$ whenever $\langle \beta, \langle c_1, \ldots, c_n \rangle, E \rangle \in \Delta_\phi$; otherwise ϕ is *context-dependent*. Almost all of the primitive functions we will encounter in the sequel will be context-independent. In fact almost all of the primitive functions we will encounter will produce constants as results, rather than arbitrary closed expressions in weak normal form.

For every rule $R \in \Delta_\phi$ of the form 8.8 we stipulate that the following judgment holds:

$$\beta \vdash \mathbf{app}(\phi, c_1, \ldots, c_n) \rightsquigarrow E.$$

In addition, we impose the following requirements:

[1] This example was inspired by a similar example in Barendregt's "The Lambda Calculus" [4], page 400.

PF1 The result of a rule is uniquely determined by the context and the arguments. Specifically, $E = E'$ whenever

$$\langle \beta, \langle c_1, \ldots, c_n \rangle, E \rangle \in \Delta_\phi \text{ and } \langle \beta, \langle c_1, \ldots, c_n \rangle, E' \rangle \in \Delta_\phi.$$

This means that Δ_ϕ can be regarded as a (partial) function that takes an assumption base and a list of constants and produces a unique closed expression in weak normal form.

PF2 The following problem is mechanically solvable: given an arbitrary assumption base β and a list of constants $c_1, \ldots, c_n$, determine whether or not Δ_ϕ is defined for β and $\langle c_1, \ldots, c_n \rangle$ (treating Δ_ϕ as a function); and if yes, produce the result (the result must be unique by the previous requirement).

The purpose of these requirements is to ensure that a primitive function (a) is indeed a function, and (b) can be mechanically implemented.

8.2.3 Semantics of special deductive forms

For each special deduction form $kwd(\Xi_1, \ldots, \Xi_k)$ of $\mathcal{L}$, a rule of the form shown below must be given, specifying the semantics of *kwd*. Each f_i function, $i = 1, \ldots, k$, produces an assumption base, while g produces a sentence; R is a relation, serving as an optional constraint that may be imposed, delimiting the applicability of the rule. We require that f_i, g, and R be computable. We write $\overrightarrow{\Pi}$ for $\beta, \Xi_1, \ldots, \Xi_k$ and $\overrightarrow{N}$ for $N_1, \ldots, N_k$.

$$\frac{f_1(\overrightarrow{\Pi}) \vdash_{\mathcal{L}} \Xi_1 \rightsquigarrow^* N_1 \quad f_2(\overrightarrow{\Pi}, N_1) \vdash_{\mathcal{L}} \Xi_2 \rightsquigarrow^* N_2 \ \cdots \ f_k(\overrightarrow{\Pi}, N_1, \ldots, N_{k-1}) \vdash_{\mathcal{L}} \Xi_k \rightsquigarrow^* N_k}{\beta \vdash_{\mathcal{L}} kwd(\Xi_1, \ldots, \Xi_k) \rightsquigarrow g(\overrightarrow{N})}$$
$$\text{provided } R(\overrightarrow{\Pi}, \overrightarrow{N}, g(\overrightarrow{N})).$$

As an example, consider again the deductive form $\mathbf{assume}(M, D)$, and suppose that the sentences of $\mathcal{L}$ are the propositions of first-order logic, denoted by P, Q, etc. Then the semantics of **assume** might be given via the rule:

$$\frac{\beta \vdash M \rightsquigarrow^* P \quad \beta \cup \{P\} \vdash D \rightsquigarrow^* Q}{\beta \vdash \mathbf{assume}(M, D) \rightsquigarrow P \Rightarrow Q}$$

Here $\overrightarrow{\Pi} = \beta, M, D$, $f_1(\overrightarrow{\Pi}) = f_1(\beta, M, D) = \beta$, $f_2(\Pi, P) = f_2(\beta, M, D, P) = \beta \cup \{P\}$, and $g(P, Q) = P \Rightarrow Q$. There are no additional constraints on the applicability of the rule, so R might be taken to be the constant "true".

8.3 Remarks

Some noteworthy points:

- Deductions and expressions are *syntactically distinct*. It is immediately evident by simple inspection whether a given phrase is an expression or a deduction. Intutively, expressions are intended to represent computations, while deductions represent logical demonstrations. Accordingly, evaluating an expression E in the context of some assumption base β might produce any result whatsoever—a number, a string, a function, or perhaps even a sentence. If E happens to produce a sentence, say a proposition P, then P could be *anything*; it might even be the negation of a proposition in β; it might even be the constant false. After all, the procedure of taking a proposition P and outputting $\neg P$ is a perfectly legitimate algorithm; I should be able to apply this algorithm to *any* given P, even one that happens to be a member of the current assumption base, and obtain the result $\neg P$. But precisely because this result is obtained from an *expression*, we make no claims about the soundness of it in relation to β. We only make such claims about the results of *deductions*.

- Functions and methods take multiple arguments rather than being curried. This is quite important for methods, because a method application is a deduction, and a deduction is always expected to return a sentence (or else fail or diverge); it cannot return any other kind of expression, such as a method. This viewpoint of a deduction D as something that you evaluate in order to obtain a *sentence* as the conclusion is central in the $\lambda\mu$-calculus. Of course this viewpoint could be preserved even if methods were unary, by packaging up all the necessary arguments of a method into a single list. However, methods would still have to return sentences, and hence currying would still not be possible in any meaningful sense.

- In a method application $\mathbf{dapp}(E, M_1, \ldots, M_k)$, the operator must be an expression E. It cannot be a deduction, because deductions can only produce sentences, and of course the value of E needs to be a method, not a sentence. An argument M_i, however, can be either an expression or a deduction. Method applications with deductions as arguments, i.e., nested deductions, play a very important role in the $\lambda\mu$-calculus: inference composition. The main mechanism for imparting an ordering on a deduction is via nested method applications. The fact that expressions, too, can be passed as arguments to methods, means that methods are higher-order: they can take arbitrary functions or methods as arguments.

- Likewise, in a function application $\mathbf{app}(E, M_1, \ldots, M_k)$ the operator must be an expression E, since its value must be a function. An argument M_i can again be either an expression or a deduction. It can be a deduction because deductions produce sentences, and a sentence can clearly be given as an argument to a function. Thus it is sensible to say, for instance, "apply the negation algorithm to the result of the deduction D", i.e., $\mathbf{app}(f, D)$, where f takes an arbitrary sentence P and outputs $\neg P$.

- Methods are abstractions of deductions. A method of the form $\mu\, I_1, \ldots, I_k \,.\, D$ is obtained from the deduction D by *abstracting* over the *parameters* $I_1, \ldots, I_k$. This abstraction mechanism is a tremendously useful tool for packaging up deductions into reusable units. Soundness is automatically ensured by the semantics of assumption bases—no type system is necessary. Every method is automatically guaranteed to produce sound conclusions. This will become more clear in due course.

- Accordingly, the body of a method must be a deduction, not an expression. Thus, something like $\mu\, x, y \,.\, \mathbf{app}(f, \mathbf{app}(g, x), \mathbf{app}(g, y))$ is nonsense at the syntactic level; it is not a well-formed phrase of the $\lambda\mu$-calculus. In contradistinction, the body of a function must be an expression, not a deduction. Thus something like $\lambda\, x \,.\, \mathbf{dapp}(f, \cdots)$ is syntactically ill-formed.

- A deduction is simply an *application* of a method—nothing else counts as a deduction.[2] A method is *not*, by itself, a deduction; it is an *expression* that captures infinitely many deductions (corresponding to the infinitely many values that the parameters may assume) in a single finite description.

8.4 Basic $\lambda\mu$ theory

We will say that a judgment $\beta \vdash_{\mathcal{L}} M \rightsquigarrow N$ is *derivable* in $\mathcal{L}$ iff there is a sequence of judgments

$$\beta_1 \vdash_{\mathcal{L}} M_1 \rightsquigarrow N_1, \ldots, \beta_k \vdash_{\mathcal{L}} M_k \rightsquigarrow N_k$$

such that $\beta_k = \beta$, $M_k = M$, $N_k = N$, and each judgment $\beta_i \vdash M_i \rightsquigarrow N_i$ is either an instance of a core axiom ([R1], [R2], or [R12]) or a δ-evaluation axiom; or else follows from previous judgments via a core rule or some special-form rule.

A phrase M is *reducible* in a given β iff there is a phrase N such that $\beta \vdash M \rightsquigarrow N$. M is *irreducible* (or in *normal form*) in β iff it is not reducible in β, i.e., iff there is no N such that $\beta \vdash M \rightsquigarrow N$. We say that M is *weakly convergent* in β iff there is a phrase N that is irreducible in β and such that $\beta \vdash M \rightsquigarrow^* N$. M is *strongly convergent* in β iff

[2] In the pure $\lambda\mu$-calculus; augmented systems also have special deductive forms.

there is no infinite sequence $M_1, M_2, \ldots, M_n, \ldots$ such that $M = M_1$ and $\beta \vdash M_i \leadsto M_{i+1}$ for all i. Clearly, strong convergence implies weak convergence. For suppose that M is strongly convergent in β. Now either M is irreducible in β or not. If it is, then $\beta \vdash M \leadsto^* M$ and M is irreducible in β, hence M is weakly convergent in β. If it is not, then there must be a phrase M_2 such that $\beta \vdash M \leadsto M_2$. Now either M_2 is irreducible in β or not. If it is, then M is weakly convergent in β; otherwise we have $\beta \vdash M_2 \leadsto M_3$ for some M_3. Eventually we must reach a phrase M_n that is irreducible in β, as otherwise M would not be strongly convergent in β. But if M_n is irreducible and $\beta \vdash M \leadsto^* M_n$ then M is weakly convergent in β. Of course weak convergence does not imply strong convergence. In addition, we say that M is *everywhere weakly convergent* iff M is weakly convergent in *every* β; and we say that M is *everywhere strongly convergent* iff it is strongly convergent in every β.

We say that M is *divergent* in β iff N is reducible in β whenever $\beta \vdash M \leadsto^* N$. The following equivalent characterization of divergence is easily derived:

Lemma 8.1 *M is divergent in β iff M is not weakly convergent in β.*

Finally, M is *everywhere* divergent iff it is divergent in every assumption base.

Theorem 8.2 *There are deductions and expressions that are everywhere divergent.*

Proof: Set

$$\Omega_{\mathbf{D}} = \mathbf{dapp}(\mu\, I\,.\,\mathbf{dapp}(I, I), \mu\, I\,.\,\mathbf{dapp}(I, I))$$

and

$$\Omega_{\mathbf{E}} = \mathbf{app}(\lambda\, I\,.\,\mathbf{app}(I, I), \lambda\, I\,.\,\mathbf{app}(I, I)).$$

Note that $\Omega_{\mathbf{D}} \in \mathit{Ded}$, $\Omega_{\mathbf{E}} \in \mathit{Exp}$. It is easy to see that for no β do we have $\beta \vdash \Omega_{\mathbf{E}} \leadsto^* N$ or $\beta \vdash \Omega_{\mathbf{D}} \leadsto^* N$ for irreducible N. ■

The following result shows that divergence (and hence weak convergence) may depend on the context. The reader must read Section 8.6.4 in order to understand the proof.

Theorem 8.3 *There are phrases which are divergent in some assumption bases and not divergent in others. Accordingly, divergence in some β does not imply divergence everywhere. Equivalently, weak convergence in some β does not imply weak convergence everywhere.*

Proof: Consider a primitive function *holds* such that

$$\beta \cup \{S\} \vdash \mathbf{app}(\mathit{holds}, S) \leadsto \mathsf{true}$$

and

$$\beta \vdash \mathbf{app}(holds, S) \leadsto \mathsf{false}$$
$$\text{whenever } S \notin \beta.$$

It is clear that *holds* satisfies **PF1** and **PF2**. Now pick any constant sentence S and let E be the expression

$$(\mathbf{cond}\ \ (holds\ \ S)\ \ S\ \ \Omega_{\mathbf{E}})$$

where $\Omega_{\mathbf{E}}$ is defined in the proof of Theorem 8.2. Clearly, $\{S\} \vdash E \leadsto S$, but E is not weakly convergent in any β that does not contain S. Hence, from Lemma 8.1, E is divergent, say, in $\emptyset$. ■

8.5 Basic $\lambda\mu$ metatheory

Let a $\lambda\mu$ system $\mathcal{L} = (kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n); C; \Delta)$ be given. A binary relation $\succcurlyeq \subseteq \mathcal{P}(Sent_C) \times Sent_C$ will be called *Tarskian* iff it is

1. *reflexive*, i.e., $\Phi \succcurlyeq S$ whenever $S \in \Phi$;
2. *monotonic*, i.e., $\Phi_1 \cup \Phi_2 \succcurlyeq S$ whenever $\Phi_1 \succcurlyeq S$; and
3. *transitive*, i.e., $\Phi \succcurlyeq S_2$ whenever $\Phi \succcurlyeq S_1$ and $\Phi \cup \{S_1\} \succcurlyeq S_2$.

Further, we say that $\succcurlyeq$ *includes* the primitive methods of $\mathcal{L}$ iff for every primitive method ψ of $\mathcal{L}$ we have $Prem(R) \succcurlyeq Con(R)$ for all $R \in \Delta_\psi$.

The evaluation semantics of the $\lambda\mu$-calculus determine a binary *deducibility relation* $\vdash_{\mathcal{L}} \subseteq \mathcal{P}(Sent_C) \times Sent_C$ as follows:

$$\Phi \vdash_{\mathcal{L}} S \text{ iff there exist } \beta \subseteq \Phi \text{ and } D \text{ such that } \beta \vdash_{\mathcal{L}} D \leadsto S.$$

If $\Phi \vdash_{\mathcal{L}} S$ we say that S is *deducible* from Φ (in $\mathcal{L}$). The *theorems* of $\mathcal{L}$ are all and only those sentences S such that $\emptyset \vdash_{\mathcal{L}} S$. When $\mathcal{L}$ is understood or irrelevant we will write $\Phi \vdash S$ instead of $\Phi \vdash_{\mathcal{L}} S$. We have:

Theorem 8.4 $\vdash_{\mathcal{L}}$ *is Tarskian.*

Proof: For reflexivity, invoke the primitive method claim (see rule [R12]). Monotonicity is immediate. Finally, transitivity follows from [R7]: If $\Phi \vdash S_1$ and $\Phi \cup \{S_1\} \vdash S_2$ then there are deductions D_1 and D_2 such that

$$\beta_1 \vdash D_1 \leadsto S_1 \tag{8.9}$$

and

$$\beta_2 \vdash D_2 \leadsto S_2 \tag{8.10}$$

for some $\beta_1 \subseteq \Phi$, $\beta_2 \subseteq \Phi \cup \{S_1\}$. Set

$$\beta = \beta_1 \cup (\beta_2 - \{S_1\})$$

so that

$$\beta_1 \subseteq \beta \tag{8.11}$$
$$\beta_2 - \{S_1\} \subseteq \beta \subseteq \Phi \tag{8.12}$$
$$\beta_2 \subseteq \beta \cup \{S_1\} \tag{8.13}$$

and

$$D = \mathbf{dapp}(\mu\, I\,.\, D_2, D_1)$$

for some I that does not occur in D_1 or in D_2, so that

$$D_2[M/I] = D_2 \tag{8.14}$$

for all phrases M. Then we have:

(a) $\beta_1 \vdash D_1 \leadsto S_1$	supposition 8.9
(b) $\beta \vdash D_1 \leadsto S_1$	(a), 8.11, [R10]
(c) $\beta_2 \vdash D_2 \leadsto S_2$	supposition 8.10
(d) $\beta \cup \{S_1\} \vdash D_2 \leadsto S_2$	(c), 8.13, [R10]
(e) $\beta \cup \{S_1\} \vdash \mathbf{dapp}(\mu\, I\,.\, D_2, S_1) \leadsto D_2$	[R2], 8.14
(f) $\beta \cup \{S_1\} \vdash \mathbf{dapp}(\mu\, I\,.\, D_2, S_1) \leadsto S_2$	(e), (d), [R11]
(g) $\beta \vdash D \leadsto S_2$	(b), (f), [R7]

Therefore, $\Phi \vdash S_2$. ■

Theorem 8.5 *If $\mathcal{L}$ is pure then $\vdash_{\mathcal{L}}$ is the least Tarskian relation that includes the primitive methods of $\mathcal{L}$.*

Proof: We have already shown that $\vdash_{\mathcal{L}}$ is Tarskian. It is also readily verified that it includes the primitive methods of $\mathcal{L}$. Specifically, pick any δ-rule

$$R = \langle \mathit{Prem}(R), \langle c_1, \ldots, c_n \rangle, \mathit{Con}(R) \rangle$$

of any primitive method ψ. Then the following evaluation axiom holds:

$$\mathit{Prem}(R) \vdash_{\mathcal{L}} \mathbf{dapp}(\psi, c_1, \ldots, c_n) \leadsto \mathit{Con}(R) \tag{8.15}$$

and, consequently, $Prem(R) \vdash_{\mathcal{L}} Con(R)$.

Lastly, we need to show that if $\succcurlyeq\ \subseteq \mathcal{P}(Sent_C) \times Sent_C$ is any Tarskian relation that includes the primitive methods of $\mathcal{L}$ then $\vdash_{\mathcal{L}}\ \subseteq\ \succcurlyeq$, i.e., $\Phi \succcurlyeq S$ whenever $\Phi \vdash_{\mathcal{L}} S$, or equivalently, that $\Phi \succcurlyeq S$ whenever $\beta \vdash D \leadsto S$ for some D and $\beta \subseteq \Phi$. We proceed by induction on the length n of the derivation of the judgment $\beta \vdash D \leadsto S$. When $n = 1$ the judgment must be an instance of a core axiom or a δ-evaluation axiom. In the first case, the only core axiom that is a possible candidate is [R12]; it could not possibly be [R1] or [R2] for syntactic reasons. Specifically, every instance of [R1] is of the form $\beta \vdash E_1 \leadsto E_2$, relating an expression to an expression, whereas in our case the judgment is of the form $\beta \vdash D \leadsto S$, taking a deduction to an expression (recall that every sentence S is a constant, and hence an expression). For similar reasons, the judgment could not be an instance of [R2], as every instance of [R2] is of the form $\beta \vdash D_1 \leadsto D_2$, relating a deduction to a deduction. Hence the judgment must be an instance of [R12], i.e., of the form $\{S\} \vdash \mathbf{dapp}(\mathsf{claim}, S) \leadsto S$, in which case $\Phi \succcurlyeq S$ follows because $\succcurlyeq$ is assumed to be Tarskian, and hence reflexive. In the second case, if the judgment is a δ-evaluation axiom of the form 8.15, the supposition that $\succcurlyeq$ includes every primitive method of $\mathcal{L}$ entails $\beta \succcurlyeq S$, and thus $\Phi \succcurlyeq S$ follows from the monotonicity of $\succcurlyeq$.

For the inductive step, suppose that the derivation is of length $n + 1$ and that the result holds for all derivations of length n or less. Then the judgment $\beta \vdash D \leadsto S$ is the last—$(n + 1)^{st}$—judgment in the derivation, and there are two possibilities as to how it is obtained: First, the judgment might again be an instance of a core axiom or a δ-evaluation axiom. In that case the result can be established as in our preceding analysis. Secondly, the judgment might follow from previous judgments in the derivation via one of the core rules of the $\lambda\mu$-calculus (Figure 8.1). We perform a case analysis: for syntactic reasons, the rule in question could not be one of [R3], [R4], [R5], [R6], [R8], or [R9], given that the obtained judgment is of the form $\beta \vdash D \leadsto c$. Hence the only remaining possibilities are [R7], [R10], and [R11]. In the case of [R7], the judgment $\beta \vdash D \leadsto S$ must be of the form $\beta \vdash \mathbf{dapp}(E, M_1, \ldots, D_i, \ldots, M_n) \leadsto S$, where the judgments $\beta \vdash D_i \leadsto S_i$ and

$$\beta \cup \{S_i\} \vdash \mathbf{dapp}(E, M_1, \ldots, S_i, \ldots, M_n) \leadsto S$$

are derivable in n or fewer steps. Accordingly, the inductive hypothesis implies $\Phi \succcurlyeq S_i$ and $\Phi \cup \{S_i\} \succcurlyeq S$, and the desired $\Phi \succcurlyeq S$ now follows from the supposition that $\succcurlyeq$ is transitive. In the case of [R10] or [R11] the result follows directly from the inductive hypothesis. This completes the case analysis and the inductive argument. ■

The *decision problem* for $\mathcal{L}$ is: given an arbitrary S, determine whether S is a theorem of $\mathcal{L}$. If this problem is mechanically solvable, we say that $\mathcal{L}$ is *decidable*. $\mathcal{L}$ is *inconsistent* iff $\emptyset \vdash_{\mathcal{L}} S$ for every S; if it is not inconsistent we call it *consistent*.

Other metatheoretical properties of $\mathcal{L}$ such as soundness and completeness are formulated with respect to some "entailment relation" $\models \subseteq \mathcal{P}(Sent_C) \times Sent_C$. Given such a relation, we define the following notions. If $\Phi \models S$ we say that Φ *entails* S. If a sentence S is entailed by the empty set we say that it is a *tautology*, or *valid* (always with respect to $\models$); this might be abbreviated as $\models S$, instead of $\emptyset \models S$. Moreover, we say that

- $\mathcal{L}$ is *sound* with respect to $\models$ iff $\vdash_{\mathcal{L}} \subseteq \models$;
- $\mathcal{L}$ is *complete* with respect to $\models$ iff $\vdash_{\mathcal{L}} \supseteq \models$;
- $\mathcal{L}$ is *tautologically complete* with respect to $\models$ iff $\emptyset \vdash_{\mathcal{L}} S$ whenever $\emptyset \models S$.

Clearly, completeness implies tautological completeness.

Theorem 8.6 (Soundness theorem, pure systems) *A pure $\lambda\mu$ system $\mathcal{L}$ is sound with respect to a Tarskian relation $\models$ iff $\models$ includes every primitive method of $\mathcal{L}$.*

Proof: One direction is trivial. The converse follows directly from Theorem 8.5. ■

Next, consider a special deductive form $kwd(\Xi_1, \ldots, \Xi_k)$, whose semantics are given by a rule of the form shown in Section 8.2.3. We say that this form *preserves* $\models$ iff $\beta \models g(\overrightarrow{N})$ whenever $\Xi_i = D$ and $f_i(\overrightarrow{P}, N_1, \ldots, N_{i-1}) \models N_i$ for $i = 1, \ldots, k$. We have:

Theorem 8.7 (Soundness theorem, augmented systems) *An augmented $\lambda\mu$ system $\mathcal{L}$ is sound with respect to a Tarskian relation $\models$ iff $\models$ includes every primitive method of $\mathcal{L}$ and is preserved by every special deductive form of $\mathcal{L}$.*

Proof: Again, one direction is immediate. For the converse, we need to show that $\Phi \models S$ whenever $\beta \vdash_{\mathcal{L}} D \leadsto S$ for some D and $\beta \subseteq \Phi$. We proceed just as in the proof of Theorem 8.5, except that there is one more possibility to consider on the inductive step: the case when the judgment $\beta \vdash D \leadsto S$ is obtained from a special-form rule (in which case, of course, D is a special deductive form). The result in that case follows from the inductive hypothesis in tandem with the supposition that the special deductive forms of $\mathcal{L}$ preserve $\models$. ■

The *entailment problem* for $\models$ is to find a strongly converging method[3] $\mu\, I \,.\, D$ such that

$$\beta \vdash \mathbf{dapp}(\mu\, I \,.\, D, S) \leadsto S \quad \text{iff} \quad \beta \models S. \tag{8.16}$$

[3]A method $\mu\, I \,.\, D$ is called strongly converging iff $\mathbf{dapp}(\mu\, I \,.\, D, M)$ is everywhere strongly convergent, for all phrases M.

In fact we are usually interested in a special case of the above problem, namely, the *validity problem* for $\models$: find a strongly converging method $\mu I \,.\, D$ such that

$$\emptyset \vdash \mathbf{dapp}(\mu I \,.\, D, S) \rightsquigarrow S \quad \text{iff} \quad \emptyset \models S, \text{i.e., iff } S \text{ is valid.} \tag{8.17}$$

Clearly, a solution to the entailment problem constitutes a solution to the validity problem.

Note that we require the relevant methods to be strongly converging to ensure that they "always terminate". Accordingly, a method which derives S from β whenever the latter entails S but diverges in all other cases does not count as a solution to the entailment problem. We want to obtain an answer in a finite amount of time, the answer being either the result S, verifying that S is indeed a consequence of β, or some kind of irreducible expression that is not a primitive value, such as $\mathbf{dapp}(\mathsf{mp}, P \wedge Q)$, indicating that an error has occured, and hence, from 8.16, that β does not entail S.

Assuming that $\models$ is compact, i.e., that $\Phi \models S$ iff $\beta \models S$ for some $\beta \subseteq \Phi$, we have:

Theorem 8.8 *If the entailment problem is solvable then $\mathcal{L}$ is complete; if, in addition, L is sound, then it is also decidable. In particular, if the validity problem is solvable and $\mathcal{L}$ is sound then $\mathcal{L}$ is tautologically complete and decidable.*

Proof: Assume there is a strongly converging method $\mu I \,.\, D$ such that

$$\beta \vdash \mathbf{dapp}(\mu I \,.\, D, S) \rightsquigarrow S$$

whenever $\beta \models S$. Pick any sentence S_0 and suppose that $\Phi \models S_0$, so that, by compactness, $\beta \models S_0$ for some $\beta \subseteq \Phi$. It follows from our assumption that

$$\beta \vdash \mathbf{dapp}(\mu I \,.\, D, S_0) \rightsquigarrow S_0$$

and hence S_0 is deducible from Φ. To decide whether an arbitrary S_0 is a theorem, simply evaluate $\mathbf{dapp}(\mu I \,.\, D, S_0)$ in $\emptyset$. By strong convergence, a result must eventually be obtained. Further, by supposition, the result will be S_0 iff S_0 is valid, i.e., by soundness and completeness, iff S_0 is a theorem. ■

Of course for many interesting systems the relation $\models$ is undecidable, and the entailment and validity problems do not have a solution. In such cases what we are interested in, from a practical standpoint, is obtaining better and better solutions of what might be termed the "automated reasoning problem" for $\models$, which is to find a strongly converging method $\mu I \,.\, D$ such that $\emptyset \vdash \mathbf{dapp}(\mu I \,.\, D, S) \rightsquigarrow S$ *for as many tautologies S as possible.*

A significant advantage of formulating these problems in terms of methods rather than functions (i.e., arbitrary algorithms) is that if $\mathcal{L}$ is sound with respect to $\models$ then the first halves of the requirements 8.16 and 8.17 are automatically satisfied: we know that if *any* deduction D produces a sentence S in an assumption base β, then β *must* entail S. Now the advantage is that, by Theorem 8.6 and Theorem 8.7, soundness is easy to verify if $\models$ is Tarskian: we simply check that the primitive methods respect $\models$ and that the special deductive forms preserve it. Then the hard work is done by the semantics of the language: soundness for the entire system $\mathcal{L}$ follows, as guaranteed by the two cited theorems, by the very design of the $\lambda\mu$-calculus. This means that simply by checking the soundness of the primitive methods we are entitled to conclude that *any* method $\mu \overrightarrow{I} . D$, no matter how complicated, even one with trillions of "lines of code" and nested recursive calls and pattern matches and so on, will never produce something that does not follow; for $\mathbf{dapp}(\mu \overrightarrow{I} . D, \overrightarrow{M})$ is, syntactically, a deduction, and the soundness theorems tell us that the result of *any* deduction in *any* β is entailed by β.

By contrast, suppose that we formulated the entailment/validity problems in terms of functions $\lambda I . E$, i.e., in terms of conventional algorithms. This is, in fact, how theorem proving is usually formulated. We want to find an algorithm, for instance, that always terminates and which produces a result S iff S is a tautology (or at any rate produces as many tautologies as possible, if the full problem is undecidable). Once we arrive at such an algorithm, however, we have the burden of proving it correct, a big part of which is proving that it never outputs something that is not a tautology, i.e., proving that it is sound. If the algorithm is sizeable and complicated, as theorem provers are bound to be, then proving its soundness will not be an easy task: long, complicated algorithms are not easily amenable to rigorous analysis. Moreover, everytime we modify our theorem prover in order to improve it we have to reprove its soundness all over again. In contradistinction, if we write our theorem prover as a method then no such analysis is ever necessary: soundness comes for free, courtesy of the $\lambda\mu$-calculus. And we can modify the method to our heart's content; it will always remain sound.

On the other hand, it is usually easier to write a theorem prover as a function than as a method. Methods require harder thinking and more work. And they are generally less efficient than functions, although that could change with more research on method optimization. But they are also more interesting to write, as they give insight into *exactly why* a result holds. A Prolog engine that is written as a predicate logic method, for instance, will perform universal instantiation on the quantified Horn clauses with the results of the unification, it will perform Modus Ponens on the instantiated clause and the (recursively proved) head goals, and so on. Essentially, then, a theorem prover written as a method will actually *prove* a given goal in full detail, it will not just "verify" it. An added advantage of this is that the prover could stack up all the primitive inferences that it performs, in the order in which they occur, and then at the end reverse the stack and offer them as a forward proof of the result. Thus theorem-proving methods have a uniform mechanism for explaining their behavior that is not available to theorem provers expressed as generic algorithms.

8.6 Conventions and syntax sugar

8.6.1 Notational conventions

For readability reasons we will often use s-expression syntax for function application, writing $(E\ M_1\ \cdots\ M_k)$ instead of $\mathbf{app}(E, M_1, \ldots, M_k)$. Parentheses will be left out whenever they are not necessary for unambiguous parsing. In particular, the outer pair of parentheses can always be omitted. Thus we might write, for instance, $\mathsf{add}\ (-\ 5\ 2)\ 7$, meaning $\mathbf{app}(\mathsf{add}, \mathbf{app}(-, 5, 2), 7)$. A similar s-expression syntax will be used for method applications, but with an exclamation mark **!** preceding the method being applied: $(\mathbf{!}E\ M_1\ \cdots\ M_k)$. This will serve to distinguish method applications from function applications. Again parentheses will often be dropped, so we might write, for example, $\mathbf{!}\mathsf{both}\ P\ (\mathbf{!}\mathsf{dn}\ \neg\neg Q)$ for

$$\mathbf{dapp}(\mathsf{both}, P, \mathbf{dapp}(\mathsf{dn}, \neg\neg Q)).$$

8.6.2 let and dlet

We introduce the following syntax sugar:

$$[\![\mathbf{dlet}\ I = M\ \mathbf{in}\ D]\!] = \mathbf{dapp}(\mu\, I\, .\, D, M) \tag{8.18}$$

and

$$[\![\mathbf{let}\ I = M\ \mathbf{in}\ E]\!] = \mathbf{app}(\lambda\, I\, .\, E, M) \tag{8.19}$$

Thus every **dlet** is a deduction, and every **let** is an expression. We may also have a **dlet** (or **let**) with multiple bindings $I_1 = M_1, \ldots, I_n = M_n$; those are desugared as shown in Fig. 8.2, with 8.18 and 8.19 as base cases.

The following lemma clarifies the sense in which **dlet**s allow for proof composition:

Lemma 8.9 *Let D be the deduction* **dlet** $I = D_1$ **in** D_2. *If*

$$\beta \vdash D_1 \leadsto S_1 \ \textit{and}\ \beta \cup \{S_1\} \vdash D_2[S_1/I] \leadsto S_2$$

then $\beta \vdash D \leadsto S_2$.

$$
[\![\begin{array}{l}\textbf{dlet } I_1 = M_1 \\ \qquad I_2 = M_2 \\ \qquad \vdots \\ \qquad I_n = M_n \\ \textbf{in} \\ \quad D\end{array}]\!] = \begin{array}{l}\textbf{dlet } I_1 = M_1 \\ \textbf{in} \\ \quad [\![\begin{array}{l}\textbf{dlet } I_2 = M_2 \\ \qquad \vdots \\ \qquad I_n = M_n \\ \textbf{in} \\ \quad D\end{array}]\!]\end{array}
\qquad
[\![\begin{array}{l}\textbf{let } I_1 = M_1 \\ \qquad I_2 = M_2 \\ \qquad \vdots \\ \qquad I_n = M_n \\ \textbf{in} \\ \quad D\end{array}]\!] = \begin{array}{l}\textbf{let } I_1 = M_1 \\ \textbf{in} \\ \quad [\![\begin{array}{l}\textbf{let } I_2 = M_2 \\ \qquad \vdots \\ \qquad I_n = M_n \\ \textbf{in} \\ \quad D\end{array}]\!]\end{array}
$$

Figure 8.2: Desugaring of **dlet**s and **let**s with multiple bindings.

Proof: By definition, D stands for $\mathbf{dapp}(\mu\, I\, .\, D_2, D_1)$, an application of a method to the result of a deduction. The key rule here is [R7]: it will entitle us to conclude $\beta \vdash D \rightsquigarrow S_2$ if we can only show $\beta \vdash D_1 \rightsquigarrow S_1$ and

$$\beta \cup \{S_1\} \vdash \mathbf{dapp}(\mu\, I\, .\, D_2, S_1) \rightsquigarrow S_2.$$

Now the former holds by supposition, while for the latter we have:

1.	$\beta \cup \{S_1\} \vdash \mathbf{dapp}(\mu\, I\, .\, D_2, S_1) \rightsquigarrow D_2[S_1/I]$	[R2]
2.	$\beta \cup \{S_1\} \vdash D_2[S_1/I] \rightsquigarrow S_2$	Supposition
3.	$\beta \cup \{S_1\} \vdash \mathbf{dapp}(\mu\, I\, .\, D_2, P) \rightsquigarrow S_2$	1, 2, [R11]

and we are now done by virtue of [R7]. ∎

The following corollary will come handy later:

Corollary 8.10 *Let* $D = \mathbf{dlet}\ I = D_1\ \mathbf{in}\ D_2$, *where* I *does not occur in* D_2. *If* $\beta \vdash D_1 \rightsquigarrow S_1$ *and* $\beta \cup \{S_1\} \vdash D_2 \rightsquigarrow S_2$ *then* $\beta \vdash D \rightsquigarrow S_2$.

8.6.3 Recursive deductions and computations

It will prove very useful to be able to formulate recursive methods of the form

$$m = \mu\, \overrightarrow{I}\, .\, \cdots\, m\, \cdots. \tag{8.20}$$

It is possible to express such methods in the $\lambda\mu$-calculus owing to the fact that methods can be the results of computations, meaning that functions can return methods. In particular, we note that any method m_0 that is a fixed point of

$$H = \lambda\, m\, .\, \mu\, \overrightarrow{I}\, .\, \cdots\, m\, \cdots \tag{8.21}$$

satisfies equation (8.20), since

$$m_0 = \mathbf{app}(H, m_0) = \mu\, \overrightarrow{I}\, .\, \cdots\ m_0\ \cdots .$$

Now a fixed point of H can be found with the regular fixed-point combinator

$$Y = \lambda\, F\, .\, \mathbf{app}(\lambda\, x\, .\, \mathbf{app}(F, \mathbf{app}(x, x)), \lambda\, x\, .\, \mathbf{app}(F, \mathbf{app}(x, x))).$$

Of course Y will not work for multi-argument functionals, but it is easy to see that it will find the fixed point of any single-argument functional H such as (8.21) (for arbitrary β):

1. $\beta \vdash \mathbf{app}(Y, H) \rightsquigarrow \mathbf{app}(\lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)), \lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)))$
2. $\beta \vdash \mathbf{app}(\lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)), \lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x))) \rightsquigarrow$
   $\mathbf{app}(H, \mathbf{app}(\lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)), \lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x))))$
3. $\beta \vdash \mathbf{app}(Y, H) \rightsquigarrow$
   $\mathbf{app}(H, \mathbf{app}(\lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)), \lambda\, x\, .\, \mathbf{app}(H, \mathbf{app}(x, x)))) = \mathbf{app}(H, \mathbf{app}(Y, H))$.

Here the first two judgments are instances of [R1], while the third judgment follows from the first two by [R11]. Thus for any β and any single-argument H we have

$$\beta \vdash \mathbf{app}(Y, H) \rightsquigarrow \mathbf{app}(H, \mathbf{app}(Y, H)) \tag{8.22}$$

or, in more conventional notation, $Y(H) = H(Y(H))$, i.e., $Y(H)$ is a fixed point of H.[4] It is the derivability of judgment 8.22 that will allow us to find solutions to equations requiring recursive methods as solutions. Of course the same idea applies to recursive functions as well. Thus we introduce the following syntax sugar: we treat

$$\mathbf{dletrec}\ I = M\ \mathbf{in}\ D$$

as an abbreviation for

$$\mathbf{dlet}\ I = \mathbf{app}(Y, \lambda\, I\, .\, M)\ \mathbf{in}\ D$$

and

$$\mathbf{letrec}\ I = M\ \mathbf{in}\ E$$

as an abbreviation for

$$\mathbf{let}\ I = \mathbf{app}(Y, \lambda\, I\, .\, M)\ \mathbf{in}\ E.$$

Thus, syntactically, a **dletrec** is a deduction while a **letrec** is an expression.

[4]Strictly speaking, the equality sign here signifies the equivalence relation generated by the symmetric closure of the quasi-order $\beta \vdash \_ \rightsquigarrow^* \_$.

8.6.4 Conditionals

In many situations it is useful to have a primitive equality function for testing whether two constants are the same. Thus the reader may assume that all of the systems we will examine in the sequel come equipped with a constant equal and two constants true and false, the first being a primitive function and the last two being primitive values. The semantics of equal are given by the δ-rules

$$\frac{}{\beta \vdash \mathbf{app}(\mathsf{equal}, c, c) \leadsto \mathsf{true}}$$

and

$$\frac{}{\beta \vdash \mathbf{app}(\mathsf{equal}, c_1, c_2) \leadsto \mathsf{false}}$$
$$\text{whenever } c_1 \neq c_2.$$

Requirement **PF2** is satisfied owing to our assumption that the equality predicate on the set of constants is decidable.

Note that in some systems true and false will also serve as sentences. In those cases, then, true and false will have dual roles as primitive values: possible results of deductions, as well as "boolean values" indicating the results of equality tests and so on.

It is useful to be able to combine conditions (phrases that return true or false) into more complex conditions through the use of boolean operators. Thus we introduce primitive functions bool-not, bool-and, and bool-or, with the expected semantics:

$$\beta \vdash \mathbf{app}(\mathsf{bool\text{-}not}, \mathsf{true}) \leadsto \mathsf{false}$$
$$\beta \vdash \mathbf{app}(\mathsf{bool\text{-}not}, \mathsf{false}) \leadsto \mathsf{true}$$
$$\beta \vdash \mathbf{app}(\mathsf{bool\text{-}and}, \mathsf{true}, \mathsf{true}) \leadsto \mathsf{true}$$

and so forth.

Next, we would like to have some means of expressing conditional computations and deductions whose outcome may depend on arbitrarily complicated conditions. In particular, we would like to express computations of the form

$$(\mathbf{cond}\ \ M\ \ E_1\ \ E_2)$$

with the following informal evaluation semantics: if the phrase M evaluates to true (in the context of some β), then evaluate E_1; otherwise, if M evaluates to false, then evaluate E_2. The case in which M produces neither true nor false is left unspeficied. Likewise, we want to be able to write deductions of the form

$$(\mathbf{dcond}\ \ M\ \ D_1\ \ D_2)$$

with similar semantics. Thus **cond** forms are to count as expressions, because regardless of whether M is true or false, $(\mathbf{cond}\ \ M\ \ E_1\ \ E_2)$ will end up evaluating an expression—either E_1 or E_2. Similarly, **dcond** forms can automatically be understood as deductions because regardless of whether M is true or false, $(\mathbf{dcond}\ \ M\ \ D_1\ \ D_2)$ will end up evaluating a deduction—either D_1 or D_2. This is important in order to maintain the distinction between computations and deductions at the syntactic level.

Although the $\lambda\mu$-calculus does not have any built-in machinery for this purpose, we can easily simulate it in terms of the primitives. The standard trick from the λ-calculus of *defining* "true" and "false" as $\lambda\, x\, .\, \lambda\, y\, .\, x$ and $\lambda\, x\, .\, \lambda\, y\, .\, y$ will not work for several reasons, as the reader will verify. But a solution can be obtained by modifying this idea. In particular, let us introduce two primitive functions if_D and if_E, whose semantics are given by the δ-rules

$$\beta \vdash \mathbf{app}(\mathsf{if}_D, \mathsf{true}) \rightsquigarrow \mu\, x, y\, .\, \mathbf{dapp}(\mathsf{claim}, x)$$
$$\beta \vdash \mathbf{app}(\mathsf{if}_D, \mathsf{false}) \rightsquigarrow \mu\, x, y\, .\, \mathbf{dapp}(\mathsf{claim}, y)$$

and

$$\beta \vdash \mathbf{app}(\mathsf{if}_E, \mathsf{true}) \rightsquigarrow \lambda\, x, y\, .\, x$$
$$\beta \vdash \mathbf{app}(\mathsf{if}_E, \mathsf{false}) \rightsquigarrow \lambda\, x, y\, .\, y.$$

We now define $(\mathbf{dcond}\ \ M\ \ D_1\ \ D_2)$ as syntax sugar for

$$\mathbf{dapp}(\mathbf{app}(\mathsf{if}_D, M), D_1, D_2)$$

and $(\mathbf{cond}\ \ M\ \ E_1\ \ E_2)$ as syntax sugar for

$$\mathbf{app}(\mathbf{app}(\mathsf{if}_E, M), E_1, E_2).$$

The reader will verify that this results in the desired semantics:

Lemma 8.11 *If* $\beta \vdash M \rightsquigarrow \mathsf{true}$ *and* $\beta \vdash D_1 \rightsquigarrow S$ *then*

$$\beta \vdash (\mathbf{dcond}\ \ M\ \ D_1\ \ D_2) \rightsquigarrow S$$

while if $\beta \vdash M \rightsquigarrow \mathsf{false}$ *and* $\beta \vdash D_2 \rightsquigarrow S$ *then*

$$\beta \vdash (\mathbf{dcond}\ \ M\ \ D_1\ \ D_2) \rightsquigarrow S.$$

Moreover, if

$$\beta \vdash M \leadsto \mathsf{true} \textit{ and } \beta \vdash E_1 \leadsto E_1'$$

then $\beta \vdash (\mathbf{cond}\ \ M\ \ E_1\ \ E_2) \leadsto E_1'$*; while if*

$$\beta \vdash M \leadsto \mathsf{false} \textit{ and } \beta \vdash E_2 \leadsto E_2'$$

then $\beta \vdash (\mathbf{cond}\ \ M\ \ E_1\ \ E_2) \leadsto E_2'$.

A $\lambda\mu$ system that contains all of the above constants and δ-rules will be called *standard*.

8.6.5 Conclusion-annotated form

With the background of the previous desugarings, we introduce the syntax

$$E \ \ \mathbf{by} \ \ D$$

as an abbreviation for the deduction

$$\mathbf{dlet}\ S = D\ \mathbf{in}\ (\mathbf{dcond}\ (\mathsf{equal}\ \ E\ \ S)\ \ (\mathbf{!}\mathsf{claim}\ \ S)\ \ D_{\mathit{fail}})$$

where D_{fail} is some fixed deduction that always fails (or, more precisely, one that is irreducible in all assumption bases). Thus the form "E **by** D" is very similar to D; if both are correct, then, denotationally, they are identical: they both produce the same result. However, "E **by** D" says something over and above D—it explicitly asserts that the results of D and E are identical. Therefore, even if D successfully produces a conclusion, the deduction "E **by** D" might still fail if the produced conclusion does not agree with E. It is this extra "documentation" that these forms provide that makes them useful. Deductions which are written in this style are said to be in *conclusion-annotated form*. It will be seen that such deductions are even more perspicuous than regular $\lambda\mu$ deductions. Accordingly, when proof *presentation* is the main concern, so that the top priorities are readability and clarity, this style is advised. When proof discovery is the main concern, this style is not necessary.

8.6.6 Pattern matching

In many $\lambda\mu$ systems it is useful to be able to *match* sentences (and perhaps other kinds of primitive values as well) against patterns. Specifically, we would like the ability to express deductions of the following abstract syntax:

$$\mathbf{dmatch}\ M\ \ \pi_1 \Longrightarrow D_1, \ldots, \pi_n \Longrightarrow D_n \tag{8.23}$$

where the various π_i are *patterns*, and having the following informal evaluation semantics: evaluate the phrase M to obtain a value, called the *discriminant value*. Then try to *match* the discriminant value against the pattern π_1. If this succeeds, evaluate D_1 in the current lexical environment augmented with the bindings resulting from the match. Otherwise try the same process with π_2 and D_2, and so on, until a successful match is found. If no match can be found, fail. We also want a similar construct on the expression (computation) level:

$$\textbf{match } M \;\; \pi_1 \Longrightarrow E_1, \ldots, \pi_n \Longrightarrow E_n. \tag{8.24}$$

In this section we will show how to express pattern matching in terms of $\lambda\mu$ primitives for one particular set of sentences: the propositions of zero-order logic. However, the same idea can be applied to other sets of sentences, and indeed, to any other domain of inductively generated compound values.

For the remainder of this section the reader should assume that the set of sentences is the set of propositions P having the following abstract syntax:

$$P \;::=\; \mathsf{true} \mid \mathsf{false} \mid A_i \mid \neg P \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P \Leftrightarrow Q$$

where $A_1, A_2, \ldots$ are propositional atoms. Patterns for these sentences can be defined via the following abstract syntax:

$$\pi \;::=\; I \mid P \mid \neg\pi \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \pi_1 \Rightarrow \pi_2 \mid \pi_1 \Leftrightarrow \pi_2$$

so that a pattern is either an identifier or a compound pattern consisting of a propositional constructor applied to a list of patterns. This basic pattern language can be enriched so as to allow for the decomposition of not only propositions but other primitive values as well, or combinations thereof (e.g., lists of propositions). The basic idea is the same and can be adapted to more complicated cases.

Again we will be simulating the desired behavior in terms of appropriate combinations of primitives: we will desugar deductions of the form 8.23 into **dlet**s and **dcond**s (which are themselves syntax sugar for method applications). Before we present the desugaring algorithm, we need to introduce a few primitive functions: "recognizer" functions such as $\mathsf{is\text{-}negation}$, $\mathsf{is\text{-}conjunction}$, $\mathsf{is\text{-}disjunction}$, etc., and "selector" functions such as $\mathsf{left\text{-}conjunct}$, $\mathsf{right\text{-}conjunct}$, $\mathsf{neg\text{-}body}$, etc. The semantics of these constants should be specified via δ-rules, which should be obvious from their names. We illustrate with conjunctions:

$$\frac{}{\beta \vdash \mathbf{app}(\mathsf{is\text{-}conjunction}, P_1 \wedge P_2) \rightsquigarrow \mathsf{true}}$$

$$\frac{}{\beta \vdash \mathbf{app}(\textsf{is-conjunction}, P) \rightsquigarrow \textsf{false}}$$
whenever P is not of the form $P_1 \wedge P_2$

$$\frac{}{\beta \vdash \mathbf{app}(\textsf{left-conjunct}, P_1 \wedge P_2) \rightsquigarrow P_1}$$

$$\frac{}{\beta \vdash \mathbf{app}(\textsf{right-conjunct}, P_1 \wedge P_2) \rightsquigarrow P_2}$$

We also assume the existence of a unary primitive function is-prop which returns true if its argument is a proposition, and false otherwise. The desugaring now proceeds as follows: a deduction of the form 8.23 becomes

$$
\begin{array}{l}
\mathbf{dlet}\ \ x_{init} = M \\
\mathbf{in} \\
\quad (\mathbf{dcond}\ \ Q_1\ \ \mathbf{dlet}\ B_1\ \mathbf{in}\ D_1 \\
\qquad\qquad\quad (\mathbf{dcond}\ \ Q_2\ \ \mathbf{dlet}\ B_2\ \mathbf{in}\ D_2 \\
\qquad\qquad\qquad\qquad\quad (\mathbf{dcond} \cdots D_{fail})))
\end{array}
$$

where D_{fail} is some fixed deduction that always fails, Q_i is an expression that will return either true or false, and B_i is a list of bindings $I_1 = E_1, \ldots, I_{k_i} = M_{k_i}$, for $i = 1, \ldots, n$. The definition of Q_i and B_i is given via the algorithm $\mathcal{T}$ below that takes a pattern π and an expression E and returns a pair (Q, B) consisting of an expression Q and a list of bindings B. In particular, we set $(Q_i, B_i) = \mathcal{T}(\pi_i, x_{init})$. The algorithm $\mathcal{T}$ is defined by induction on the structure of the pattern π:

$$
\begin{array}{l}
\mathcal{T}(\pi, E) = \mathit{match}\ \pi \\
\qquad\qquad\quad \mathrm{I}\ \rightarrow (\mathbf{app}(\textsf{is-prop}, E), [I = E]) \\
\qquad\qquad\quad \neg\pi\ \rightarrow \mathit{let}\ \ (Q, B) = \mathcal{T}(\pi, \mathbf{app}(\textsf{neg-body}, E))\ \ \mathit{in} \\
\qquad\qquad\qquad\quad (\mathbf{app}(\textsf{bool-and}, \mathbf{app}(\textsf{is-negation}, E), Q), B) \\
\qquad\qquad\quad \pi_1 \wedge \pi_2\ \rightarrow \mathit{let}\ \ (Q_1, B_1) = \mathcal{T}(\pi_1, \mathbf{app}(\textsf{left-and}, E)) \\
\qquad\qquad\qquad\qquad\qquad (Q_2, B_2) = \mathcal{T}(\pi_2, \mathbf{app}(\textsf{right-and}, E))\ \ \mathit{in} \\
\qquad\qquad\qquad\qquad (\mathbf{app}(\textsf{bool-and}, \mathbf{app}(\textsf{bool-and}, \mathbf{app}(\textsf{is-conjunction}, E), Q_1), Q_2), \\
\qquad\qquad\qquad\qquad\ \mathit{join}(B_1, B_2)) \\
\qquad\qquad\quad \pi_1 \vee \pi_2\ \rightarrow \mathit{let}\ \ (Q_1, B_1) = \mathcal{T}(\pi_1, \mathbf{app}(\textsf{left-or}, E)) \\
\qquad\qquad\qquad\qquad\qquad (Q_2, B_2) = \mathcal{T}(\pi_2, \mathbf{app}(\textsf{right-or}, E))\ \ \mathit{in} \\
\qquad\qquad\qquad\qquad (\mathbf{app}(\textsf{bool-and}, \mathbf{app}(\textsf{bool-and}, \mathbf{app}(\textsf{is-disjunction}, E), Q_1), Q_2), \\
\qquad\qquad\qquad\qquad\ \mathit{join}(B_1, B_2))
\end{array}
$$

where we write *join* for list concatenation. The remaining two cases (conditional and biconditional patterns) are treated similarly to conjunctions and disjunctions.

A successful match *decomposes* (or "destructures") the discriminant value in accordance with the structure of the pattern. The identifiers of the pattern act as variables that get bound to appropriate parts of the discriminant value. Note that the identifiers of the pattern π_i (see 8.23) introduce lexical *scope*: their scope is the deduction D_i. If D_i itself contains another **dmatch** deduction with a pattern that uses one of the identifiers I of π_i, as in

$$
\begin{array}{l}
\textbf{dmatch}\ \ P \\
\quad P_1 \wedge P_2 \implies \textbf{dmatch}\ \ P_2 \\
\qquad\qquad\qquad\qquad P_3 \vee (P_1 \wedge P_5) \implies \cdots
\end{array}
$$

then a new binding takes effect for I, masking the outer one from π_i and introducing new scope. Our desugaring handles these issues automatically by binding pattern identifiers in properly nested **dlet**s.

The desugaring of expressions of the form 8.24 is entirely analogous to that given above.

8.7 Interpreting the $\lambda\mu$-calculus

In this section we will be concerned with the task of reducing a phrase M, in a given assumption base β, to a *weakly irreducible* expression, i.e., to an expression in weak normal form, which will be regarded as "the result" of M in β. We will say that a phrase M is *normalizable* in β iff there is a weakly irreducible phrase N such that $\beta \vdash M \leadsto^* N$. It is clear that normalizibility in β implies neither strong nor weak convergence in β (take $\lambda\, x\, .\, \Omega_{\mathbf{E}}$ as a counter-example). The converse does not hold either, however, even for closed phrases (for open phrases it clearly does not hold, e.g., any identifier I by itself is strongly convergent in every assumption base but not normalizable). For example, if no δ-rules apply to a primitive-function application of the form $\mathbf{app}(\phi, c_1, \ldots, c_n)$, then the latter expression is strongly convergent but not normalizable.

By an "interpreter" for the $\lambda\mu$-calculus we will mean an algorithm that takes as input a phrase M and an an assumption base β and attempts to "normalize" M, i.e., transform it into a phrase N in weak normal form such that $\beta \vdash M \leadsto^* N$. Thus an interpeter might be viewed as a semi-decision procedure (albeit an incomplete one, as we will see soon) for constructively answering the question "Is a phrase M normalizable in β?", where a positive answer is always accompanied by producing a particular weakly irreducible expression N such that $\beta \vdash M \leadsto^* N$. The algorithm might result in an error in attempting to evaluate an application of the form $\mathbf{app}(\phi, E_1, \ldots, E_n)$ or $\mathbf{dapp}(\psi, E_1, \ldots, E_n)$ where each E_i is in normal form and there are no applicable

$$
\begin{array}{l}
\textit{eval-args}\ [M_1, \ldots, M_k]\ \beta\ \textit{Eval} = \textit{eval}^*\ [M_1, \ldots, M_k]\ []\ \emptyset \\
\text{where} \\
\quad \textit{eval}^*\ []\ V\ \Phi = (\textit{rev}(V), \Phi) \\
\quad \textit{eval}^*\ [E, M_2, \ldots, M_k]\ V\ \Phi = \\
\qquad\qquad \textit{let}\ N = \textit{Eval}(E, \beta) \\
\qquad\qquad \textit{in} \\
\qquad\qquad\quad \textit{eval}^*\ [M_2, \ldots, M_k]\ N{::}V\ \Phi \\
\quad \textit{eval}^*\ [D, M_2, \ldots, M_k]\ V\ \Phi = \\
\qquad\qquad \textit{let}\ S = \textit{Eval}(D, \beta) \\
\qquad\qquad \textit{in} \\
\qquad\qquad\quad \textit{eval}^*\ [M_2, \ldots, M_k]\ S{::}V\ \Phi \cup \{S\}
\end{array}
$$

Figure 8.3: An argument-evaluation algorithm, parameterized over *Eval*.

δ-rules for ϕ, ψ (this can be decided effectively owing to **PF2** and **PM2**). An error might also occur if the input phrase is not closed. Of course the algorithm might also diverge. The gist of the evaluation process is the reduction of λs and μs via rules [R1] and [R2], and the reduction of primitive method and function applications via δ-rules, as in the regular λ-calculus. However, there is an extra complication arising from rule [R7] that we will discuss shortly.

A *normal-order* evaluation strategy always contracts the left-most, outer-most λ or μ redex first, whereas an *applicative-order* (eager) evaluation strategy contracts the left-most, inner-most redex. In what follows we will present three interpreters for the $\lambda\mu$-calculus. The first two will be substitution-based, the first using a normal-order evaluation strategy and the second using an applicative-order strategy. The third algorithm will be based on environments and closures rather than substitution, and will also use an eager strategy.

We assume the availability of a function *eval-prim* which takes a constant c, a list of expressions $E_1, \ldots, E_n$ in weak normal form, and an assumption base β, and returns the result of applying c to $E_1, \ldots, E_n$ in β (assuming that c is either a primitive function or a primitive method). Presumably, $\textit{eval-prim}(c, [E_1, \ldots, E_n], \beta)$ will produce a result c' iff a judgment of the form

$$\beta \vdash \mathbf{app}(c, E_1, \ldots, E_n) \rightsquigarrow c'$$

or

$$\beta \vdash \mathbf{dapp}(c, E_1, \ldots, E_n) \rightsquigarrow c'$$

is a δ-evaluation axiom. If that is not the case, or if c is not a primitive method or function, or if the number or form of arguments is wrong, then *eval-prim* should generate an error. This specification of *eval-prim* is realizable owing to the requirements we imposed on primitive functions and methods in Section 8.2.

The algorithms are presented in functional notation similar to that used in Chapter 5. The normal-order interpreter is shown in Figure 8.4, and the applicative-order one in Figure 8.5. They both use an argument-evaluation algorithm *eval-args*, shown in Figure 8.3, which takes a list of phrases $[M_1, \ldots, M_k]$ to be evaluated (representing the argument list of a function or method application), along with an assumption base β in which to evaluate them and an evaluator *Eval* to perform the evaluation (note that we write *rev* for list reversal). Barring error or non-termination, a call *eval-args* $[M_1, \ldots, M_k]$ β *Eval* will return a pair $([v_1, \ldots, v_k], \Phi)$ consisting of a list of "values" $[v_1, \ldots, v_k]$ (these will be expressions in weak normal form representing the evaluation results for the inputs $M_1, \ldots, M_k$) and a set $\Phi \subseteq \{v_1, \ldots, v_k\}$ that contains the value v_i iff the phrase M_i is a deduction. Accordingly, Φ may be regarded as the set of lemmas or intermediate conclusions established prior to the application. Keeping track of Φ is necessary in order to handle rule [R7] without having to resort to backtracking. Thus if $M_1, \ldots, M_k$ are the arguments of a method application, then after evaluating them we can proceed in a forward manner and apply the method in the context $\beta \cup \Phi$, rather than just β, thereby incorporating into the assumption base whatever sentences were deductively derived. The idea is that if the application can go through in $\beta \cup \beta'$, where β' is *any* subset of the values of the deductive arguments, then, by monotonicity, it will certainly go through in $\beta \cup \Phi$, since $\Phi \supseteq \beta'$. Hence there is no need to backtrack and try some other subset in case of failure. We need only try the application once, using the largest possible β': Φ. Of course in practice every member of Φ is usually needed; an element of Φ that is not necessary for the method application would constitute an extraneous lemma.

Moreover, in the case of augmented systems, for each special deductive form

$$kwd(\Xi_1, \ldots, \Xi_k)$$

with semantics given by a rule of the form shown in Section 8.2.3, the call-by-name interpreter should have a clause of the following form:

$$
\begin{aligned}
&\mathit{Eval}_N(kwd(M_1, \ldots, M_k), \beta) = \\
&\qquad \mathit{let}\quad v_1 = \mathit{Eval}_N(M_1, f_1(\beta, M_1, \ldots, M_k)) \\
&\qquad \phantom{\mathit{let}\quad} v_2 = \mathit{Eval}_N(M_2, f_2(\beta, M_1, \ldots, M_k, v_1)) \\
&\qquad \phantom{\mathit{let}\quad v_2 =} \vdots \\
&\qquad \phantom{\mathit{let}\quad} v_k = \mathit{Eval}_N(M_k, f_k(\beta, M_1, \ldots, M_k, v_1, \ldots, v_{k-1}))
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{Eval}_N(c, \beta) = c \\
\mathit{Eval}_N(I, \beta) = \mathit{fail} \ \text{“Free identifier”} \\
\mathit{Eval}_N(\lambda \overrightarrow{I} \,.\, E, \beta) = \lambda \overrightarrow{I} \,.\, E \\
\mathit{Eval}_N(\mu \overrightarrow{I} \,.\, D, \beta) = \mu \overrightarrow{I} \,.\, D \\
\mathit{Eval}_N(\mathbf{app}(E, M_1, \ldots, M_k), \beta) = \\
\qquad \mathit{match}\ \ \mathit{Eval}_N(E, \beta) \\
\qquad\quad \lambda \overrightarrow{I} \,.\, E_b \longrightarrow \ \mathit{Eval}_N(E_b[M_1, \ldots, M_k / I_1, \ldots, I_k], \beta) \\
\qquad\quad c \longrightarrow \ \mathit{let}\ \ ([v_1, \ldots, v_k], \Phi) = \mathit{eval\text{-}args}\ \ [M_1, \ldots, M_k]\ \ \beta\ \ \mathit{Eval}_N \\
\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad \mathit{eval\text{-}prim}(c, [v_1, \ldots, v_k], \beta) \\
\qquad\quad \_ \longrightarrow \mathit{fail}\ \ \text{“Invalid function application”} \\
\mathit{Eval}_N(\mathbf{dapp}(E, M_1, \ldots, M_k), \beta) = \\
\qquad \mathit{match}\ \ \mathit{Eval}_N(E, \beta) \\
\qquad\quad \mu \overrightarrow{I} \,.\, D \longrightarrow \ \mathit{Eval}_N(D[M_1, \ldots, M_k / I_1, \ldots, I_k], \beta) \\
\qquad\quad c \longrightarrow \ \mathit{let}\ \ ([v_1, \ldots, v_k], \Phi) = \mathit{eval\text{-}args}\ \ [M_1, \ldots, M_k]\ \ \beta\ \ \mathit{Eval}_N \\
\qquad\qquad\quad \mathit{in} \\
\qquad\qquad\qquad \mathit{eval\text{-}prim}(c, [v_1, \ldots, v_k], \beta \cup \Phi) \\
\qquad\quad \_ \longrightarrow \mathit{fail}\ \ \text{“Invalid method application”}
\end{array}
$$

Figure 8.4: A normal-order substitution-based interpreter for the $\lambda\mu$-calculus.

$$
\begin{array}{l}
\qquad v \ \ = g(v_1, \ldots, v_k) \\
\mathit{in} \\
\quad R(\beta, M_1, \ldots, M_k, v_1, \ldots, v_k, v) \rightarrow v, \mathit{fail}
\end{array}
$$

Entirely analogous clauses should appear in the applicative-order interpreter.

The final $\lambda\mu$ interpeter we present, Eval_C, is based on environments and closures. It brings to the surface certain idiosyncracies of the $\lambda\mu$-calculus that pass unnoticed in the substitution-based interpreters. By an environment ρ we will mean a finite function from identifiers to values, where a value is either a constant or a closure, and where a function (method) *closure* is a triple containing a list of identifiers, called the *parameters* of the closure; a phrase, called the *body* of the closure (this will be an expression for function closures and a deduction for method closures); and an environment ρ, called the *closure environment*. We write $\mathit{fclos}([I_1, \ldots, I_k], E, \rho)$

$$
\begin{array}{l}
Eval_A(c,\beta) = c \\
Eval_A(I,\beta) = \mathit{fail} \text{ ``Free identifier''} \\
Eval_A(\lambda \overrightarrow{I} \,.\, E,\beta) = \lambda \overrightarrow{I} \,.\, E \\
Eval_A(\mu \overrightarrow{I} \,.\, D,\beta) = \mu \overrightarrow{I} \,.\, D \\
Eval_A(\mathbf{app}(E, M_1, \ldots, M_k),\beta) = \\
\qquad \mathit{let}\ ([v_1,\ldots,v_k],\Phi) = \mathit{eval\text{-}args}\ [M_1,\ldots,M_k]\ \beta\ Eval_A \\
\qquad \mathit{in} \\
\qquad\quad \mathit{match}\ Eval_A(E,\beta) \\
\qquad\qquad \lambda \overrightarrow{I} \,.\, E_b \longrightarrow Eval_A(E_b[v_1,\ldots,v_k/I_1,\ldots,I_k],\beta) \\
\qquad\qquad c \longrightarrow \mathit{eval\text{-}prim}(c,[v_1,\ldots,v_k],\beta) \\
\qquad\qquad \_ \longrightarrow \mathit{fail} \text{ ``Invalid function application''} \\
Eval_A(\mathbf{dapp}(E, M_1, \ldots, M_k),\beta) = \\
\qquad \mathit{let}\ ([v_1,\ldots,v_k],\Phi) = \mathit{eval\text{-}args}\ [M_1,\ldots,M_k]\ \beta\ Eval_A \\
\qquad \mathit{in} \\
\qquad\quad \mathit{match}\ Eval_A(E,\beta) \\
\qquad\qquad \mu \overrightarrow{I} \,.\, D \longrightarrow Eval_A(D[v_1,\ldots,v_k/I_1,\ldots,I_k],\beta \cup \Phi) \\
\qquad\qquad c \longrightarrow \mathit{eval\text{-}prim}(c,[v_1,\ldots,v_k],\beta \cup \Phi) \\
\qquad\qquad \_ \longrightarrow \mathit{fail} \text{ ``Invalid method application''}
\end{array}
$$

Figure 8.5: An applicative-order substitution-based interpreter for the $\lambda\mu$-calculus.

($\mathit{mclos}([I_1,\ldots,I_k], D, \rho)$) to designate a function (method) closure with parameters $I_1,\ldots,I_k$, body E (D), and environment ρ; and we write $\rho[I_1 \mapsto v_1,\ldots,I_k \mapsto v_k]$ for the environment that maps I_j to v_j and every other I to $\rho(I)$ (we assume $I_1,\ldots,I_k$ are distinct).

$Eval_C$ takes a phrase, an environment, and an assumption base, and produces a value (either a constant or a closure), or else it reports an error or gets into an infinite loop. The definition of $Eval_C(M,\rho,\beta)$ is given by structural recursion on M in Figure 8.6. Note that according to our conventions about δ-rules, a primitive function application may produce not only a constant, but also a closed expression of the form $\lambda \overrightarrow{I} \,.\, E$ or $\mu \overrightarrow{I} \,.\, D$. Hence, to adhere to our above specifications (of always returning either a constant or a closure), the interpreter must "close" such an expression, i.e., turn it into the closure $\mathit{fclos}(\overrightarrow{I}, E, \emptyset)$ or $\mathit{mclos}(\overrightarrow{I}, D, \emptyset)$, where $\emptyset$ is the empty environment. Accordingly, the interpreter applies a *close* function on the result of a primitive

$$
\begin{array}{l}
\mathit{Eval}_C(c, \rho, \beta) = c \\
\mathit{Eval}_C(I, \rho, \beta) = \rho(I) \\
\mathit{Eval}_C(\lambda \overrightarrow{I} \,.\, E, \rho, \beta) = \mathit{fclos}(\overrightarrow{I}, E, \rho) \\
\mathit{Eval}_C(\mu \overrightarrow{I} \,.\, D, \rho, \beta) = \mathit{mclos}(\overrightarrow{I}, D, \rho) \\
\mathit{Eval}_C(\mathbf{app}(E, M_1, \ldots, M_k), \rho, \beta) = \\
\qquad\qquad \mathit{let}\ \ ([v_1, \ldots, v_k], \Phi) = \mathit{eval\text{-}args\text{-}env}\ [M_1, \ldots, M_k]\ \ \rho\ \ \beta \\
\qquad\qquad \mathit{in} \\
\qquad\qquad\quad \mathit{match}\ \ \mathit{Eval}_C(E, \rho, \beta) \\
\qquad\qquad\qquad \mathit{fclos}([I_1, \ldots, I_k], E_b, \rho') \longrightarrow\ \mathit{Eval}_C(E_b, \rho'[I_1 \mapsto v_1, \ldots, I_k \mapsto v_k], \beta) \\
\qquad\qquad\qquad c \longrightarrow\ \mathit{close}(\mathit{eval\text{-}prim}(c, [v_1, \ldots, v_k], \beta)) \\
\qquad\qquad\qquad \_ \longrightarrow \mathit{fail}\ \text{“Invalid function application”} \\
\mathit{Eval}_C(\mathbf{dapp}(E, M_1, \ldots, M_k), \rho, \beta) = \\
\qquad\qquad \mathit{let}\ \ ([v_1, \ldots, v_k], \Phi) = \mathit{eval\text{-}args\text{-}env}\ [M_1, \ldots, M_k]\ \ \rho\ \ \beta \\
\qquad\qquad \mathit{in} \\
\qquad\qquad\quad \mathit{match}\ \ \mathit{Eval}_C(E, \rho, \beta) \\
\qquad\qquad\qquad \mathit{mclos}([I_1, \ldots, I_k], D, \rho') \longrightarrow\ \mathit{Eval}_C(D, \rho'[I_1 \mapsto v_1, \ldots, I_k \mapsto v_k], \beta \cup \Phi) \\
\qquad\qquad\qquad c \longrightarrow\ \mathit{eval\text{-}prim}(c, [v_1, \ldots, v_k], \beta \cup \Phi) \\
\qquad\qquad\qquad \_ \longrightarrow \mathit{fail}\ \text{“Invalid method application”} \\
\text{and} \\
\mathit{eval\text{-}args\text{-}env}\ [M_1, \ldots, M_k]\ \ \rho\ \ \beta = \mathit{eval}^*\ \ [M_1, \ldots, M_k]\ \ []\ \ \emptyset \\
\text{where} \\
\quad \mathit{eval}^*\ \ []\ \ V\ \ \Phi = (\mathit{rev}(V), \Phi) \\
\quad \mathit{eval}^*\ \ [E, M_2, \ldots, M_k]\ \ V\ \ \Phi = \\
\qquad\qquad \mathit{let}\ \ N = \mathit{Eval}_C(E, \rho, \beta) \\
\qquad\qquad \mathit{in} \\
\qquad\qquad\quad \mathit{eval}^*\ \ [M_2, \ldots, M_k]\ \ N{::}V\ \ \Phi \\
\quad \mathit{eval}^*\ \ [D, M_2, \ldots, M_k]\ \ V\ \ \Phi = \\
\qquad\qquad \mathit{let}\ \ S = \mathit{Eval}_C(D, \rho, \beta) \\
\qquad\qquad \mathit{in} \\
\qquad\qquad\quad \mathit{eval}^*\ \ [M_2, \ldots, M_k]\ \ S{::}V\ \ \Phi \cup \{S\}
\end{array}
$$

Figure 8.6: An applicative-order environment-based interpreter for the $\lambda\mu$-calculus.

function application. This function returns a constant unchanged, while it “closes” an expression $\lambda \overrightarrow{I} \,.\, E$ or $\mu \overrightarrow{I} \,.\, D$ in the manner indicated above.

The important points to notice, which are not salient in the substitution-based interpreters, are the following:

1. When a method closure is *evaluated*, the assumption base in which the evaluation takes place is thrown away; only the lexical environment is retained in the closure. This reflects the intuition that what we know at the point at which a method is formed is immaterial; what matters is the state of the world *at the point at which the method is applied.* Accordingly:

2. When a method closure is *applied* to a list of values, the body of the closure is evaluated *in the current assumption base*; the assumption base in which the method was formed has been forgotten, as we explained above. Thus we see that methods are *statically scoped* with respect to lexical environments, but *dynamically scoped* with respect to assumption bases.

These points apply to function closures as well, but that is immaterial if computation and deduction are sequestered so that expressions never access the assumption base. In fact if all the primitive functions are context-independent, as is almost always the case, then the fact that function bodies are dynamically scoped with respect to assumption bases is completely inconsequential: function bodies may be evaluated in the current assumption base, but that will not be of any significance if expressions are oblivious to the contents of the assumption base. Thus in such cases it makes no difference whether function closure bodies are evaluated with respect to the current assumption base, the static assumption base (the assumption base in which the closure was formed), or even the empty assumption base. However, the issue becomes relevant when context-dependent primitives are provided (such as the function *holds* in the proof of Theorem 8.3), as might be done in order to enable users to write theorem provers as functions rather than methods. We will discuss this further in the sequel.

We close by discussing the correctness and completeness of the interpreters.

Theorem 8.12 *If $Eval_N(M, \beta) = N$ then N is in weak normal form, and further, $\beta \vdash M \leadsto^* N$. Likewise, if $Eval_A(M, \beta) = N$ then N is in weak normal form and $\beta \vdash M \leadsto^* N$.*

Regarding completeness, we observe that none of the three interpeters is guaranteed to find a weak normal form for the input phrase, even if one exists. This is in contrast to the regular λ-calculus, where a normal-order reduction strategy is guaranteed to converge to a weak normal form (and even a regular normal form) if one exists. The reason why this does not hold in the $\lambda\mu$ setting is rule [R7]: consider a method application having two deductions D_1 and D_2 as arguments, and suppose that (a) the application will go through if both D_1 and D_2 go through, (b) D_1 goes through on the supposition that the result of D_2 holds *but not otherwise*, and (c) D_2 goes through. By virtue of [R7], this means that the entire application should be successful: we can evaluate D_2 first, then D_1, and then the overall application. But if our strategy is to evaluate arguments left-to-right, then D_1 will clearly fail, and so will the overall deduction. As an example, let the application be

$$\mathbf{dapp}(\mathsf{both}, \mathbf{dapp}(\mathsf{mp}, P \Rightarrow Q, P), \mathbf{dapp}(\mathsf{dn}, \neg\neg P))$$

in the assumption base $\{\neg\neg P, P \Rightarrow Q\}$, where dn and mp stand for double negation and modus ponens, respectively, with the obvious δ-rules (see Figure 9.2). Clearly, this deduction will fail with a left-to-right argument evaluation strategy, even though the $\lambda\mu$ semantics can successfully reduce it to the sentence $Q \wedge P$. In fact it is easy to see that this situation arises with *any* fixed argument-evaluation strategy, be it left-to-right, right-to-left, or anything else—even a parallel one. There will always be a temporal permutation of the arguments that would enable the deduction to succeed but runs counter to the evaluation order.

From a practical standpoint the issue is inconsequential. Just as languages such as Scheme or ML adopt an applicative-order strategy even though this might fail to accord with the formal semantics of the λ-calculus, in the same manner it is perfectly sensible for an implementation of the $\lambda\mu$-calculus to adopt a fixed argument-evaluation order for methods even if this might fail to accord with the formal semantics. In fact making an argument deduction contingent on the result of a fellow argument deduction is patently bad style and very unlikely to be encountered in practice.

8.8 A simple example

In this section we will consider a pure $\lambda\mu$ system $Parity = (C, \Delta)$ for proving that a number is even or odd. The set of primitive values Val_C contains all strings generated by the grammar $n ::= \mathsf{0} \mid \mathsf{s}(n)$ (hereafter these strings will be called simply "numbers"); as well as the constants Even and Odd, which will be called "parity signs". As primitive values we also take all pairs of the form $\langle n, p\rangle$, where n is a number and p a parity sign. These will be the *sentences* of the system. For readability, a sentence of the form $\langle n, p\rangle$ will be written as $n : p$, e.g., $\mathsf{s}(\mathsf{0}) : \mathsf{Odd}$. Hence, evaluating deductions in this system will result in "theorems" of the form $n : p$, asserting that "n is p".

$Meth_C$ contains three primitive methods: zero-axiom, make-even, and make-odd. The associated δ-rules are given by the following schemas:

$$\emptyset \vdash \mathbf{dapp}(\mathsf{zero\text{-}axiom}) \rightsquigarrow \mathsf{0} : \mathsf{Even}$$
$$\{n : \mathsf{Odd}\} \vdash \mathbf{dapp}(\mathsf{make\text{-}even}, n : \mathsf{Odd}) \rightsquigarrow \mathsf{s}(n) : \mathsf{Even}$$
$$\{n : \mathsf{Even}\} \vdash \mathbf{dapp}(\mathsf{make\text{-}odd}, n : \mathsf{Even}) \rightsquigarrow \mathsf{s}(n) : \mathsf{Odd}$$

1.	$\emptyset \vdash \mathbf{dapp}(\mathsf{za}) \rightsquigarrow \mathsf{0} : \mathsf{Even}$	δ-rule
2.	$\{\mathsf{0} : \mathsf{Even}\} \vdash \mathbf{dapp}(\mathsf{mo}, \mathsf{0} : \mathsf{Even}) \rightsquigarrow \mathsf{s}(\mathsf{0}) : \mathsf{Odd}$	δ-rule
3.	$\emptyset \vdash \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{za})) \rightsquigarrow \mathsf{s}(\mathsf{0}) : \mathsf{Odd}$	1, 2, [R7]
4.	$\{\mathsf{s}(\mathsf{0}) : \mathsf{Odd}\} \vdash \mathbf{dapp}(\mathsf{me}, \mathsf{s}(\mathsf{0}) : \mathsf{Odd}) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{0})) : \mathsf{Even}$	δ-rule
5.	$\emptyset \vdash \mathbf{dapp}(\mathsf{me}, \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{za}))) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{0})) : \mathsf{Even}$	3, 4, [R7]
6.	$\{\mathsf{s}(\mathsf{s}(\mathsf{0})) : \mathsf{Even}\} \vdash \mathbf{dapp}(\mathsf{mo}, \mathsf{s}(\mathsf{s}(\mathsf{0})) : \mathsf{Even}) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) : \mathsf{Odd}$	δ-rule
7.	$\emptyset \vdash \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{me}, \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{za})))) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) : \mathsf{Odd}$	5, 6, [R7]
8.	$\{\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) : \mathsf{Odd}\} \vdash \mathbf{dapp}(\mathsf{me}, \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) : \mathsf{Odd}) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) : \mathsf{Even}$	δ-rule
9.	$\emptyset \vdash \mathbf{dapp}(\mathsf{me}, \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{me}, \mathbf{dapp}(\mathsf{mo}, \mathbf{dapp}(\mathsf{za}))))) \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) : \mathsf{Even}$	7, 8, [R7]

Figure 8.7: The derivation of a judgment of the form $\beta \vdash D \rightsquigarrow S$. Note the crucial role played by [R7].

Note that all three methods are uniform and compositional. In more conventional notation, the above rules might be depicted as:

$$\frac{}{\mathsf{0} : \mathsf{Even}} \qquad \frac{n : \mathsf{Odd}}{\mathsf{s}(n) : \mathsf{Even}} \qquad \frac{n : \mathsf{Even}}{\mathsf{s}(n) : \mathsf{Odd}}$$

Finally, we will assume that the system is standard, i.e., that it includes all the constants and δ-rules introduced in Section 8.6.4. This completes the specification of the system.

The following is a deduction D that proves the sentence $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) : \mathsf{Even}$:

$$\mathbf{dapp}(\mathsf{make\text{-}even}, \mathbf{dapp}(\mathsf{make\text{-}odd}, \mathbf{dapp}(\mathsf{make\text{-}even}, \mathbf{dapp}(\mathsf{make\text{-}odd}, \mathbf{dapp}(\mathsf{zero\text{-}axiom}))))) \quad (8.25)$$

Using the notational conventions of Section 8.6.1, D is written as follows:

$$\mathbf{!}\mathsf{make\text{-}even}\ \ (\mathbf{!}\mathsf{make\text{-}odd}\ \ (\mathbf{!}\mathsf{make\text{-}even}\ \ (\mathbf{!}\mathsf{make\text{-}odd}\ \ (\mathbf{!}\mathsf{zero\text{-}axiom})))).$$

The derivation shown in Figure 8.7 establishes the judgment

$$\emptyset \vdash D \rightsquigarrow \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) : \mathsf{Even}$$

(where in order to save space we write za, mo, and me in place of zero-axiom, make-odd, and make-even, respectively).

Here it is as a **dlet**:

$$
\begin{array}{ll}
\textbf{dlet} & S_0 = !\mathsf{zero\text{-}axiom} \\
& S_1 = !\mathsf{make\text{-}odd}\ \ S_0 \\
& S_2 = !\mathsf{make\text{-}even}\ \ S_1 \\
& S_3 = !\mathsf{make\text{-}odd}\ \ S_2 \\
\textbf{in} & \\
\ \ !\mathsf{make\text{-}even}\ \ S_3 &
\end{array}
$$

Note that if one desugars this **dlet** and reduces the resulting μ-redexes the obtained deduction is precisely 8.25. Finally, here is the above **dlet** expressed in conclusion-annotated form:

$$
\begin{array}{ll}
\textbf{dlet} & S_0 = \mathsf{0} : \mathsf{Even}\ \textbf{by}\ !\mathsf{zero\text{-}axiom} \\
& S_1 = \mathsf{s}(\mathsf{0}) : \mathsf{Odd}\ \textbf{by}\ !\mathsf{make\text{-}odd}\ \ S_0 \\
& S_2 = \mathsf{s}(\mathsf{s}(\mathsf{0})) : \mathsf{Even}\ \textbf{by}\ !\mathsf{make\text{-}even}\ \ S_1 \\
& S_3 = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) : \mathsf{Odd}\ \textbf{by}\ !\mathsf{make\text{-}odd}\ \ S_2 \\
\textbf{in} & \\
\ \ \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) : \mathsf{Even}\ \textbf{by}\ !\mathsf{make\text{-}even}\ \ S_3 &
\end{array}
$$

The gains in readability appear substantial when one considers the starting point 8.25.

Next, let us say that a sentence $n : \mathsf{Even}$ is *true* iff $M[\![n]\!]$ is even, where $M[\![\mathsf{0}]\!] = 0$, $M[\![\mathsf{s}(n)]\!] = 1 + M[\![n]\!]$; while a sentence $n : \mathsf{Odd}$ is true iff $M[\![n]\!]$ is odd. We now define an entailment relation $\beta \models S$ as follows: S is true whenever every element of β is true. For instance, we have

$$\{\mathsf{0} : \mathsf{Odd}\} \models \mathsf{s}(\mathsf{0}) : \mathsf{Even}.$$

The following two lemmas are immediate:

Lemma 8.13 $\models$ *is Tarskian.*

Lemma 8.14 $\emptyset \models S$ *iff* S *is true. In other words,* S *is valid iff* S *is true.*

Lemma 8.15 $\models$ *includes all three primitive methods.*

Proof: By straightforward inspection. For $\mathsf{zero\text{-}axiom}$ we have $\emptyset \models \mathsf{0} : \mathsf{Even}$, since $\mathsf{0} : \mathsf{Even}$ is true. For $\mathsf{make\text{-}odd}$, we clearly have $\{n : \mathsf{Even}\} \models \mathsf{s}(n) : \mathsf{Odd}$ for all n. And for $\mathsf{make\text{-}even}$, we have $\{n : \mathsf{Odd}\} \models \mathsf{s}(n) : \mathsf{Even}$. ■

Thus we derive the soundness of the system:

Theorem 8.16 $\vdash_{Parity}$ *is sound with respect to* $\models$*. That is, if* $\beta \vdash_{Parity} D \rightsquigarrow S$ *then* $\beta \models S$*.In particular, all theorems of Parity are true, i.e.,*

$$\models S \text{ whenever } \emptyset \vdash_{Parity} D \rightsquigarrow S.$$

Proof: Immediate from Lemma 8.13, Lemma 8.15, and the Soundness Theorem for pure systems (Theorem 8.6). ∎

Corollary 8.17 *Parity is consistent.*

Proof: The sentence $\mathsf{0} : \mathsf{Odd}$ cannot be provable because it is not true: by soundness, only true sentences are theorems. ∎

We now turn our attention to the subject of *computation as proof*, or perhaps more appropriately, computation as *theorem proving*. Specifically, we want to write a method *find-parity* that will take an arbitrary number n as input and will derive a sentence of the form $n : p$, where the sign p represents the parity of n. The main advantage of doing this with a method rather than with a function is guaranteed correctness. Because a method call is, syntactically, a deduction, and because we have proven—easily, thanks to Theorem 8.6—that all deductions produce sound results, we can rest assured that if the call **dapp**(*find-parity*, n) returns a result $n : p$, that result will be correct. Of course in the case of computing parity signs the issue is rather trivial, but our goal here is only to illustrate the general principles; for other problems the advantage of guaranteed correctness is of considerable practical importance (see our treatment of unification in Section 9.6, for instance). With correctness guaranteed then, all we have to do is make sure that the method will always return *some* sentence of the form $n : p$. This more or less boils down to verifying that the method always terminates, something that we would also have to do for a function anyway (over and above proving that the function produces correct results).

Another advantage of methods is that they can shed light on many interesting logical properties of the system. In particular, if we manage to write a method that can derive all valid sentences then we may automatically conclude that the system is (tautologically) complete, i.e., that every valid sentence is derivable. Observe that the same is *not* the case for functions, i.e., for conventional algorithms. Generally speaking, finding an algorithm that generates all valid sentences is of little value: it will not by itself entitle us to conclude anything about the completeness of the logic, for the mere fact that an algorithm produces a sentence S does not necessarily mean that S is derivable (to take an extreme case, an algorithm which generates all sentences trivially generates all valid sentences; but we cannot conclude from this that the latter are in fact provable). By contrast, if a method generates a sentence S then we *know* that S is provable in the logic. Therefore, the mere fact that the method can generate every valid sentence suffices to establish that the logic is complete. In fact the method can be taken as *constructive* proof that the logic is complete: we are not merely told that there is a derivation for any given tautology, but we are in fact given a recipe of how to construct such a derivation.[5]

Moreover, if we manage to solve the validity problem, i.e., discover a method $\mu I \,.\, D$ that always terminates and such that $\mathbf{dapp}(\mu I \,.\, D, S)$ produces S iff S is valid, then not only do we get completeness as a corollary, we also get a decision procedure for theoremhood/validity (the two will coincide by soundness and completeness): given any S, apply the method to it. If we eventually get S as a result, we know that S is a tautology/theorem; if we do not get S back, we know that it is not. Hence we have proved, again constructively, that the logic is decidable. As one might expect, it is usually the case that if we discover a method that derives all valid sentences then it becomes relatively easy to solve the validity problem. Of course for rich logics these tasks will be quite difficult, if not impossible. For instance, solving the validity problem even for propositional logic is challenging. For first-order logic it is impossible. For incomplete logics it is not even an issue. We are simply claiming that, whenever it is applicable, the approach of methods has considerable merits.

In the sequel these themes will resurface in different settings. The logics will be different, but the underlying ideas will be the same. For example, in the "arithmetic calculus" of Section 9.2 we show how to perform numerical computation with methods, by proving theorems. And we show that the underlying logic is complete and decidable by presenting a method that derives every valid sentence.

We continue with the method *find-parity*, which is to take an arbitrary n and derive its parity in the form of a theorem $n : p$. For convenience, we will assume we have patterns for matching numbers, parity signs, and sentences. For instance, number patterns π could be generated by the grammar $\pi ::= \mathsf{0} \mid I \mid \mathsf{s}(\pi)$, parity signs patterns η by the grammar $\eta ::= \mathsf{Odd} \mid \mathsf{Even} \mid I$, and sentence patterns σ by $\sigma ::= \pi : \eta$. We could then consider deductions with abstract syntax

$$\begin{gathered}\mathbf{dmatch}\ M\ \ \pi_1 \Longrightarrow D_1, \ldots, \pi_n \Longrightarrow D_n,\\ \mathbf{dmatch}\ M\ \ \eta_1 \Longrightarrow D_1, \ldots, \eta_n \Longrightarrow D_n, \text{ and}\\ \mathbf{dmatch}\ M\ \ \sigma_1 \Longrightarrow D_1, \ldots, \sigma_n \Longrightarrow D_n\end{gathered}$$

for matching against numbers, parity signs, and sentences respectively. Of course we could also have expressions of the form

$$\mathbf{match}\ M\ \ \pi_1 \Longrightarrow E_1, \ldots, \pi_n \Longrightarrow E_n$$

[5] A plain algorithm could do the same, of course, by outputting a proof of S along with each produced sentence S, and then completeness would indeed follow if we could show that every valid S is thus produced. But there is an important difference: we would *also* have to show that every proof that the algorithm could ever produce is sound, which would require a challenging analysis of the algorithm—and would also be ad hoc. With methods we get this guarantee automatically and in a uniform fashion, across a wide spectrum of different logics.

etc. The same method that we presented in Section 8.6.6 could be used to desugar such deductions and expressions into primitive $\lambda\mu$ syntax. From now on we will not bother to define pattern languages in detail. We will use patterns "as needed" in a somewhat ad hoc manner, but it will always be straightforward to determine their intended meaning from the context. The precise desugaring of **dmatch** deductions and **match** expressions will always be achievable using the techniques we discussed in Section 8.6.6; the details will be left to the reader who is interested in working them out.

We can now write *find-parity* as follows:

$$
\begin{array}{l}
\textit{find-parity} = \mu\, n\,.\\
\quad \textbf{dmatch}\;\; n\\
\qquad \mathsf{0} \Longrightarrow (!\mathsf{zero\text{-}axiom})\\
\qquad \mathsf{s}(k) \Longrightarrow \textbf{dlet}\;\; S = !\textit{find-parity}\;\; k\\
\qquad\qquad\qquad \textbf{in}\\
\qquad\qquad\qquad\quad \textbf{dmatch}\;\; S\\
\qquad\qquad\qquad\qquad k : \mathsf{Even} \Longrightarrow !\mathsf{make\text{-}odd}\;\; S\\
\qquad\qquad\qquad\qquad k : \mathsf{Odd} \Longrightarrow !\mathsf{make\text{-}even}\;\; S
\end{array}
$$

A simple structural induction on n will show[6] that the method call $!\textit{find-parity}\;\; n$ will always return some sentence of the form $n : p$. In combination with the system's soundness, we get:

Lemma 8.18 *For every number n, evaluating the call $(!\textit{find-parity}\;\; n)$ in the empty assumption base produces a true sentence of the form $n : p$.*

Of course this means that *find-parity* generates every valid sentence, so we already know that the logic is complete. We can also show completeness constructively by slightly modifying the above method to solve the validity problem:

$$
\begin{array}{l}
\textit{prove-theorem} = \mu\, S\,.\\
\quad \textbf{dmatch}\;\; S\\
\qquad n : p \Longrightarrow !\textit{find-parity}\;\; n
\end{array}
$$

[6] In fact if we actually desugared the body of *find-parity* into pure $\lambda\mu$-calculus, the required argument would be far from simple. But then again if we desugared the usual definition of the factorial function into pure λ-calculus, "simple" inductive facts about it would no longer be simple either. In essence, then, when we need to reason about a $\lambda\mu$ method it is reasonable to imagine that the method is not desugared but rather written in a language that directly offers all the relevant constructs (recursion, conditionals, matching, etc.), and that these constructs have the expected semantics. Using those hypothetical semantics directly, then, it would in principle be fairly straightforward to verify simple properties such as the above. In fact in a realistic situation that would be the case: in practice one would not write methods in a $\lambda\mu$ language that desugared everything into abstractions and applications.

With the aid of the foregoing lemma, it is easy to see that this method solves the validity problem: it always halts, and given any sentence S, the call (**!** *prove-theorem* S) will result in S iff S is valid. We conclude:

Theorem 8.19 *The system Parity is consistent, sound, complete, decidable, and its validity problem is solvable.*

Captain Willard: They told me that you had gone totally insane and that your methods were unsound.
Colonel Kurtz: Are my methods unsound?
Captain Willard: I don't see ... any method ... at all.

Apocalypse now

9 Examples of $\lambda\mu$ systems

9.1 $\lambda\mu$–H_0, a Hilbert calculus

In this section we define $\lambda\mu$–H_0, a $\lambda\mu$ system for a Hilbert-style propositional logic. The sentences of $\lambda\mu$–H_0 are the propositions generated by the abstract grammar:

$$P \ ::= \ A_i \mid \neg P \mid P \Rightarrow Q$$

where $A_1, A_2, \ldots$ are distinct propositional atoms. We use the letters P and Q to denote such sentences. There are four primitive methods: mp, K, S, and N. Their δ-rules are as follows:

$$\begin{gathered}
\emptyset \vdash \mathbf{dapp}(\mathsf{K}, P, Q) \rightsquigarrow P \Rightarrow (Q \Rightarrow P) \\
\emptyset \vdash \mathbf{dapp}(\mathsf{S}, P, Q, R) \rightsquigarrow [P \Rightarrow (Q \Rightarrow R)] \Rightarrow [(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)] \\
\emptyset \vdash \mathbf{dapp}(\mathsf{N}, Q, P) \rightsquigarrow (\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q) \\
\{P \Rightarrow Q, P\} \vdash \mathbf{dapp}(\mathsf{mp}, P \Rightarrow Q, P) \rightsquigarrow Q
\end{gathered}$$

Hereooo is a deduction D that derives $P \Rightarrow R$ from $\{P \Rightarrow Q, Q \Rightarrow R\}$:

$$\begin{aligned}
&\mathbf{dapp}(\mathsf{mp},\mathbf{dapp}(\mathsf{mp},\mathbf{dapp}(\mathsf{S},\ P, Q, R), \\
&\qquad\qquad\qquad\qquad \mathbf{dapp}(\mathsf{mp},\mathbf{dapp}(\mathsf{K},Q \Rightarrow R, P), \\
&\qquad\qquad\qquad\qquad\qquad\qquad Q \Rightarrow R)), \\
&\qquad\quad P \Rightarrow Q)
\end{aligned} \tag{9.1}$$

The reader will verify that

$$\{P \Rightarrow Q, Q \Rightarrow R\} \vdash_{\lambda\mu\text{–}H_0} D \rightsquigarrow P \Rightarrow R.$$

The preceding deduction is not very readable of course. Here is another version that uses **dlet**; note that this **dlet** desugars exactly into the above **dapp**:

$$
\begin{array}{l}
\textbf{dlet}\ S_1 = \textbf{dapp}(\mathsf{K}, Q \Rightarrow R, P) \\
\quad\quad S_2 = \textbf{dapp}(\mathsf{mp}, S_1, Q \Rightarrow R) \\
\quad\quad S_3 = \textbf{dapp}(\mathsf{S}, P, Q, R) \\
\quad\quad S_4 = \textbf{dapp}(\mathsf{mp}, S_3, S_2) \\
\textbf{in} \\
\quad \textbf{dapp}(\mathsf{mp}, S_4, P \Rightarrow Q)
\end{array}
$$

or, using the convention of writing $(!E\ \ M_1 \cdots M_n)$ for $\textbf{dapp}(E, M_1, \ldots, M_n)$:

$$
\begin{array}{l}
\textbf{dlet}\ S_1 = !\mathsf{K}\ \ Q \Rightarrow R\ \ P \\
\quad\quad S_2 = !\mathsf{mp}\ \ S_1\ \ Q \Rightarrow R \\
\quad\quad S_3 = !\mathsf{S}\ \ P\ \ Q\ \ R \\
\quad\quad S_4 = !\mathsf{mp}\ \ S_3\ \ S_2 \\
\textbf{in} \\
\quad !\mathsf{mp}\ \ S_4\ \ P \Rightarrow Q
\end{array}
$$

And in conclusion-annotated form:

$$
\begin{array}{l}
\textbf{dlet}\ S_1 = (Q \Rightarrow R) \Rightarrow [P \Rightarrow (Q \Rightarrow R)]\ \textbf{by}\ !\mathsf{K}\ \ Q \Rightarrow R\ \ P \\
\quad\quad S_2 = P \Rightarrow (Q \Rightarrow R)\ \textbf{by}\ !\mathsf{mp}\ \ S_1\ \ Q \Rightarrow R \\
\quad\quad S_3 = [P \Rightarrow (Q \Rightarrow R)] \Rightarrow [(P \Rightarrow Q) \Rightarrow (P \Rightarrow R)]\ \textbf{by}\ !\mathsf{S}\ \ P\ \ Q\ \ R \\
\quad\quad S_4 = (P \Rightarrow Q) \Rightarrow (P \Rightarrow R)\ \textbf{by}\ !\mathsf{mp}\ \ S_3\ \ S_2 \\
\textbf{in} \\
\quad P \Rightarrow R\ \textbf{by}\ !\mathsf{mp}\ \ S_4\ \ P \Rightarrow Q
\end{array}
$$

The reader should compare the readability of this deduction against that of the starting point 9.1.

As another example, here is a deduction D such that $\emptyset \vdash D \rightsquigarrow P \Rightarrow P$, for arbitrary P:

$$
\begin{array}{l}
\textbf{dapp}(\mathsf{mp},\textbf{dapp}(\mathsf{mp},\ \textbf{dapp}(\mathsf{S},P, P \Rightarrow P, P), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{dapp}(\mathsf{K},P, P \Rightarrow P)) \\
\quad\quad\quad\quad \textbf{dapp}(\mathsf{K},P, P))
\end{array}
$$

Here it is as a **dlet**:

dlet $S_1 = !\mathsf{S}\ \ P\ \ P \Rightarrow P\ \ P$
$S_2 = !\mathsf{K}\ \ P\ \ P \Rightarrow P$
$S_3 = !\mathsf{mp}\ \ S_1\ \ S_2$
$S_4 = !\mathsf{K}\ \ P\ \ P$
in
$!\mathsf{mp}\ \ S_3\ \ S_3$

And in conclusion-annotated form:

dlet $S_1 = [P \Rightarrow ((P \Rightarrow P) \Rightarrow P)] \Rightarrow [(P \Rightarrow (P \Rightarrow P)) \Rightarrow (P \Rightarrow P)]$ **by** $!\mathsf{S}\ \ P\ \ P \Rightarrow P\ \ P$
$S_2 = P \Rightarrow ((P \Rightarrow P) \Rightarrow P)$ **by** $!\mathsf{K}\ \ P\ \ P \Rightarrow P$
$S_3 = (P \Rightarrow (P \Rightarrow P)) \Rightarrow (P \Rightarrow P)$ **by** $!\mathsf{mp}\ \ S_1\ \ S_2$
$S_4 = P \Rightarrow (P \Rightarrow P)$ **by** $!\mathsf{K}\ \ P\ \ P$
in
$P \Rightarrow P$ **by** $!\mathsf{mp}\ \ S_3\ \ S_3$

The logical entailment relation $\models$ from assumption bases to propositions is defined as usual. Proving $\lambda\mu$–H_0 sound is easy, by Theorem 8.6, since we only need to show that $\models$ contains every primitive method of $\lambda\mu$–H_0, which is straightforward.

Theorem 9.1 *$\lambda\mu$–H_0 is sound: if $\beta \vdash D \rightsquigarrow P$ then $\beta \models P$.*

The system is also complete, but we will not prove this here since it is not germane to our present concerns.

9.2 The arithmetic calculus: numerical computation as theorem proving

In this section we present a pure $\lambda\mu$ system $\mathit{Arith} = (C, \Delta)$ for deducing ground equations in elementary number theory. The set of primitive values $\mathit{Val}_C \subseteq C$ includes all the strings generated by the grammar

$$e \ ::= \ \mathsf{0} \mid \mathsf{s}(e) \mid (e_1 + e_2) \mid (e_1 * e_2) \mid \mathsf{fact}(e).$$

These strings will be called "expressions". We will use the letter e to range over expressions. As a convention, when writing expressions we will omit the outer pair of parentheses, e.g., writing $e_1 * e_2$ rather that $(e_1 * e_2)$.

We single out some expressions as *numerals*. Intuitively, numerals can be regarded as normal forms for expressions. Specifically, a numeral is either the expression $\mathsf{0}$ or an expression of the form $\mathsf{s}(x)$, where x is a numeral. We will use the letters x, y, and z to range over numerals.

As primitive values we also take all pairs of the form $\langle e_1, e_2\rangle$, where e_1 and e_2 are arbitrary expressions. A pair of this form will be called an *equation*, and it will be more suggestively written as $e_1 = e_2$. These will be the *sentences* of *Arith*. Accordingly, assumption bases in this system will contain equations; and the result of a deduction will be an equation. An equation will be said to be in *solved form* iff it is of the form $e = x$, for arbitrary expression e and numeral x. Intuitively, an equation in solved form $e = x$ asserts that the "value" of e is x. There are no other primitive values outside expressions, equations, and the "standard" values described in Section 8.6.4 (true, false, etc.).

The following constants are the primitive methods of *Arith*: plus-1, plus-2, times-1, times-2, fact-1, fact-2, ref, tran, sym, s-cong, $+$-cong, $*$-cong, and fact-cong. Their semantics are given by the δ-rule schemes shown in Figure 9.1. The rules for plus-1, plus-2, times-1, times-2, fact-1, and fact-2 amount to the usual primitive recursive definitions of the addition, multiplication, and factorial functions, respectively, though written somewhat idiosyncratically. Primitive methods suffixed by -1 constitute the "base case" of the corresponding definition, while those ending in -2 constitute the inductive case. In more conventional notation, for instance, the rules for plus-1 and plus-2 may be written as follows:

$$\frac{}{e + 0 = e} \qquad \frac{e_1 + e_2 = e_3}{e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)}$$

The methods ref, sym, and tran model equational reflexivity, symmetry, and transitivity, respectively. Finally, the methods ending in -cong are congruence rules. For example, in more traditional notation s-cong and $*$-cong may be written as:

$$\frac{e = e'}{\mathsf{s}(e) = \mathsf{s}(e')} \qquad \frac{e_1 = e_1' \quad e_2 = e_2'}{e_1 * e_2 = e_1' * e_2'}$$

This completes the specification of the system.

Here is a deduction D, in conclusion-annotated form, proving that

$$\mathsf{s}(\mathsf{s}(0)) * \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$$

(more precisely, we have $\emptyset \vdash D \rightsquigarrow \mathsf{s}(\mathsf{s}(0)) * \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$):

dlet $I_1 = \mathsf{s}(\mathsf{s}(0)) * 0 = 0$ **by** !times-1 $\mathsf{s}(\mathsf{s}(0))$
$I_2 = \mathsf{s}(\mathsf{s}(0)) + 0 = \mathsf{s}(\mathsf{s}(0))$ **by** !plus-1 $\mathsf{s}(\mathsf{s}(0))$
$I_3 = \mathsf{s}(\mathsf{s}(0)) * \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0))$ **by** !times-2 I_1 I_2

$$\emptyset \vdash \mathbf{dapp}(\mathsf{plus\text{-}1}, e) \rightsquigarrow e + 0 = e$$
$$\{e_1 + e_2 = e_3\} \vdash \mathbf{dapp}(\mathsf{plus\text{-}2}, e_1 + e_2 = e_3) \rightsquigarrow e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{times\text{-}1}, e) \rightsquigarrow e * 0 = 0$$
$$\{e_1 * e_2 = e_3,\ e_1 + e_3 = e_4\} \vdash \mathbf{dapp}(\mathsf{times\text{-}2}, e_1 * e_2 = e_3, e_1 + e_3 = e_4) \vdash \rightsquigarrow e_1 * \mathsf{s}(e_2) = e_4$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{fact\text{-}1}) \rightsquigarrow \mathsf{fact}(0) = \mathsf{s}(0)$$
$$\{\mathsf{fact}(e_1) = e_2, \mathsf{s}(e_1) * e_2 = e_3\} \vdash \mathbf{dapp}(\mathsf{fact\text{-}2}, \mathsf{fact}(e_1) = e_2, \mathsf{s}(e_1) * e_2 = e_3) \rightsquigarrow \mathsf{fact}(\mathsf{s}(e_1)) = e_3$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{ref}, e) \rightsquigarrow e = e$$
$$\{e_1 = e_2\} \vdash \mathbf{dapp}(\mathsf{sym}, e_1 = e_2) \rightsquigarrow e_2 = e_1$$
$$\{e_1 = e_2, e_2 = e_3\} \vdash \mathbf{dapp}(\mathsf{tran}, e_1 = e_2, e_2 = e_3) \rightsquigarrow e_1 = e_3$$
$$\{e = e'\} \vdash \mathbf{dapp}(\mathsf{s\text{-}cong}, e = e') \rightsquigarrow \mathsf{s}(e) = \mathsf{s}(e')$$
$$\{e_1 = e_1', e_2 = e_2'\} \vdash \mathbf{dapp}(+\text{-}\mathsf{cong}, e_1 = e_1', e_2 = e_2') \rightsquigarrow (e_1 + e_2) = (e_1' + e_2')$$
$$\{e_1 = e_1', e_2 = e_2'\} \vdash \mathbf{dapp}(*\text{-}\mathsf{cong}, e_1 = e_1', e_2 = e_2') \rightsquigarrow (e_1 * e_2) = (e_1' * e_2')$$
$$\{e = e'\} \vdash \mathbf{dapp}(\mathsf{fact\text{-}cong}, e = e') \rightsquigarrow \mathsf{fact}(e) = \mathsf{fact}(e')$$

Figure 9.1: δ-rules for the primitive methods.

$I_4 = \mathsf{s}(\mathsf{s}(0)) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$ **by** !plus-2 I_2
$I_5 = \mathsf{s}(\mathsf{s}(0)) + \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$ **by** !plus-2 I_4
in
$\mathsf{s}(\mathsf{s}(0)) * \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$ **by** !times-2 I_3 I_5

The set of all expressions is clearly a freely generated term algebra, and there is a unique epimorphism V from it to the algebra formed by the natural numbers N under the corresponding constants and operations (namely, the constant 0 and the successor, addition, multiplication, and factorial operations) that extends the empty mapping $V_0 : \emptyset \longrightarrow \emptyset$, so that

$$\begin{array}{rcl} V[\![0]\!] & = & 0 \\ V[\![\mathsf{s}(e)]\!] & = & V[\![e]\!] + 1 \\ V[\![(e_1 + e_2)]\!] & = & V[\![e_1]\!] + V[\![e_1]\!] \\ V[\![(e_1 * e_2)]\!] & = & V[\![e_1]\!] \cdot V[\![e_1]\!] \\ V[\![\mathsf{fact}(e)]\!] & = & V[\![e]\!]!. \end{array}$$

We refer to the number $V[\![e]\!]$ as *the value* of e.

The mapping V will be the center of the system's semantics. Specifically, we will say that an equation $e_1 = e_2$ is *true* iff $V[\![e_1]\!] = V[\![e_2]\!]$. For instance, the equations

$\mathsf{0} = \mathsf{0}$ and $\mathsf{fact}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) = \mathsf{s}(\mathsf{s}(\mathsf{0})) * \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))$ are true, but $\mathsf{0} = \mathsf{fact}(\mathsf{s}(\mathsf{0}))$ is not. Further, for any set of equations β and single equation S, we define $\beta \models S$ to hold iff S is true whenever every equation in β is true. The following results are immediate:

Lemma 9.2 $\models$ *is Tarskian.*

Lemma 9.3 $\models e_1 = e_2$ *iff* $e_1 = e_2$ *is true. That is, the valid sentences are all and only the true equations.*

Lemma 9.4 $\models$ *includes every primitive method.*

Proof: By straightforward inspection of the δ-rules of every primitive method. We illustrate with $\mathsf{plus\text{-}2}$: we need to show that $\mathit{Prem}(R) \models \mathit{Con}R$ for every δ-rule R of $\mathsf{plus\text{-}2}$, namely, that

$$\{e_1 + e_2 = e_3\} \models e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$$

for arbitrary expressions e_1, e_2, e_3, i.e., that $e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$ is true on the supposition that $e_1 + e_2 = e_3$ is true. On that supposition it follows, by definition of truth, that $V[\![e_1 + e_2]\!] = V[\![e_3]\!]$, and by definition of V, that $V[\![e_1]\!] + V[\![e_2]\!] = V[\![e_3]\!]$. Therefore,

$$V[\![e_1]\!] + V[\![e_2]\!] + 1 = V[\![e_3]\!] + 1$$

hence

$$V[\![e_1]\!] + V[\![\mathsf{s}(e_2)]\!] = V[\![\mathsf{s}(e_3)]\!]$$

hence

$$V[\![e_1 + \mathsf{s}(e_2)]\!] = V[\![\mathsf{s}(e_3)]\!]$$

which is to say that the equation $e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$ is true. ∎

Thus we get:

Theorem 9.5 (Soundness) $\vdash_{\mathit{Arith}}$ *is sound with respect to* $\models$*, i.e.,* $\beta \models e_1 = e_2$ *whenever* $\beta \vdash_{\mathit{Arith}} e_1 = e_2$*. In particular,* $\models e_1 = e_2$ *whenever* $\emptyset \vdash_{\mathit{Arith}} e_1 = e_2$*, which is to say that all theorems of Arith are true.*

Proof: Immediate from the preceding lemmas and Theorem 8.6. ∎

Corollary 9.6 *Arith is consistent.*

Proof: The equation $0 = \mathsf{s}(0)$ is not a theorem. If it were, then, by soundness, it would have to be true. ■

We now turn to the subject of computing as theorem proving. First we consider the following problem: given two arbitrary numerals x and y, find their sum. This can be viewed as a theorem proving problem in the following light: given x and y, prove an equation of the form $x + y = z$, where the numeral z represents the sum of x and y. Now the task is to write a *method* that does this for arbitrary x and y. As we explained in the *Parity* example (page 308), the advantage of computing with methods is guaranteed correctness. In our case, Theorem 9.5 guarantees that if a method returns an equation of the form $x + y = z$, then z will indeed denote the sum of x and y. (Of course there is nothing special about arithmetic operations. By applying the same viewpoint to an arbitrary function $f(x_1, \ldots, x_n)$, one can regard *all* computation as proof search: to compute $f(a_1, \ldots, a_n)$ for given arguments $a_1, \ldots, a_n$, prove a theorem of the form $f(a_1, \ldots, a_n) = a$, using axioms that define f.) Now here is a method for adding numerals:

$add = \mu\, x, y\,.$
  dmatch y
    $0 \Longrightarrow\ !\mathsf{plus\text{-}1}\ x$
    $\mathsf{s}(u) \Longrightarrow\ !\mathsf{plus\text{-}2}\ (!\,add\ x\ u)$

As an example, the reader should convince himself that

$$\emptyset \vdash !\,add\ \mathsf{s}(0)\ \ \mathsf{s}(\mathsf{s}(0)) \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(\mathsf{s}(0))).$$

A simple induction on y will show that for all numerals x and y, the method call $(!\,add\ x\ y)$ will result in a sentence of the form $x + y = z$, for some numeral z. In combination with the soundness of *Arith*, we get:

Lemma 9.7 *For all numerals x and y, the method call $(!\,add\ x\ y)$ results in the equation $x + y = z$, where the value of the numeral z is the sum of the values of x and y.*

Before continuing we should stress that the compositionality of primitive methods has a large impact on the readability and writability of deductions and methods. When primitive methods are compositional, it is possible for one method call to appear directly as an argument to another method call, so that the result of one inference becomes a premise of another inference. This "threading" is intuitive, enhances the perspicuity of deductions, and facilitates the writing of methods. As an example, consider the primitive method $\mathsf{plus\text{-}2}$. In every δ-rule $\langle\{S_1\}, \langle S_2\rangle, S_3\rangle$ of $\mathsf{plus\text{-}2}$, the argument S_2 and the premise S_1 are identical: an equation of the form $e_1 + e_2 = e_3$. Given such an equation as an argument, the method checks that the equation is in the assumption base, and if so, it produces the output sentence $S_3 : e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$. That is what happens, evaluation-wise, when $\mathsf{plus\text{-}2}$ is applied to an argument (note here the importance of requirement **PM2**). Thus it is possible for a deduction D that produces an equation of the form $e_1 + e_2 = e_3$ to appear directly as the argument of $\mathsf{plus\text{-}2}$: $\mathbf{dapp}(\mathsf{plus\text{-}2}, D)$. By rule [R7], by the time we come to apply $\mathsf{plus\text{-}2}$ to the result of D, the latter will be incorporated in the assumption base, where $\mathsf{plus\text{-}2}$ will then find it as expected. Indeed, this is precisely what happens in the method call $\mathbf{!}\mathsf{plus\text{-}2}\ \ (\mathbf{!}\mathit{add}\ \ x\ \ u)$ in the body of *add*.

Now to get a feeling for what things would look like without compositionality, let's say that we specified the δ-rules of $\mathsf{plus\text{-}2}$ as follows:

$$\{e_1 + e_2 = e_3\} \vdash \mathbf{dapp}(\mathsf{plus\text{-}2}, e_1, e_2, e_3) \rightsquigarrow e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$$

so that $\mathsf{plus\text{-}2}$ is now a ternary method and *not* compositional. In principle, this set of δ-rules is acceptable because it satisfies **PM1** and **PM2**. To see that the latter is satisfied, in particular, think of what is involved at evaluation-time in applying $\mathsf{plus\text{-}2}$ to given arguments e_1, e_2, e_3: checking that the relevant premise is in the assumption base, and generating the correct output. Now observe that the given list of arguments e_1, e_2, e_3 enables us to carry out both: it unambiguously determines what premise we should look for in the assumption base, namely, $e_1 + e_2 = e_3$; and it unambiguously determines what sentence we should output, namely, $e_1 + \mathsf{s}(e_2) = \mathsf{s}(e_3)$ (note, incidentally, that neither would be unambiguously determined if the method received a single *set* of arguments $\{a_1, a_2, a_3\}$; the order is important). Thus this version of $\mathsf{plus\text{-}2}$ is acceptable, and is in fact extensionally equivalent with the preceding version in terms of premises and conclusions. However, the non-compositionality makes the rule very awkward to use. As an example, the reader should try to write the *add* method with this new version of $\mathsf{plus\text{-}2}$. It would look something like this:

```
add = μ x, y .
  dmatch y
    0 ⟹ !plus-1 x
    s(u) ⟹ dlet eq = !add x u
                  plus-term = lhs eq
                  e1 = left-plus plus-term
                  e2 = right-plus plus-term
                  result = rhs eq
             in
               !plus-2 e1 e2 result
```

where *left-plus* is a function that extracts the term e_1 from an expression of the form $e_1 + e_2$ (and symmetrically for *right-plus*); while *lhs* ("left-hand side") is a function that takes an equation $e_1 = e_2$ as input and returns e_1 (and likewise for *rhs*). If you think that this is bad, check to see what the methods *mult*, *factorial*, and *Eval*, given below, would look like if the relevant primitive methods were not compositional. As methods get more and more complex, the code would get more and more jumbled. The heuristic principle that emerges from this discussion: whenever possible, primitive methods should be made compositional.

We continue with a multiplication method *mult*, which takes two numerals x and y and proves an equation of the form $x * y = z$, where z is a numeral that represents the product of x and y. We will use the simple auxiliary function *rhs* mentioned above, which takes an equation $e_1 = e_2$ and returns e_2; *rhs* can be straightforwardly implemented using pattern matching:

$$
\begin{array}{l}
\mathit{rhs} = \lambda\, \mathit{eq}\,. \\
\quad \mathbf{match}\ \ \mathit{eq} \\
\qquad e_1 = e_2 \implies e_2
\end{array}
$$

(keep in mind that equations are constants and can be freely passed around to functions and methods). We can now write *mult* as follows:

$$
\begin{array}{l}
\mathit{mult} = \mu\, x, y\,. \\
\quad \mathbf{dmatch}\ \ y \\
\qquad \mathsf{0} \implies \mathbf{!}\mathsf{times\text{-}1}\ \ x \\
\qquad \mathsf{s}(u) \implies \mathbf{dlet}\ \ \mathit{eq1} = \mathbf{!}\,\mathit{mult}\ \ x\ \ u \\
\qquad\qquad\qquad\qquad\quad \mathit{eq2} = \mathbf{!}\,\mathit{add}\ \ x\ \ (\mathit{rhs}\ \ \mathit{eq1}) \\
\qquad\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{times\text{-}2}\ \ \mathit{eq1}\ \ \mathit{eq2}
\end{array}
$$

As an example, the reader should verify that

$$\emptyset \vdash \mathbf{!}\,\mathit{mult}\ \ \mathsf{s}(\mathsf{s}(\mathsf{0}))\ \ \mathsf{s}(\mathsf{s}(\mathsf{0})) \leadsto \mathsf{s}(\mathsf{s}(\mathsf{0})) * \mathsf{s}(\mathsf{s}(\mathsf{0})) = \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))).$$

As with *add*, we conclude:

Lemma 9.8 *For all numerals x and y, the method call ($\mathbf{!}\,\mathit{mult}\ x\ y$) results in the equation $x * y = z$, where the value of the numeral z is the product of the values of x and y.*

We continue with a method *factorial* that takes a numeral x and derives an equation $\mathsf{fact}(x) = y$ for a numeral y that represents the factorial of x:

$$
\begin{array}{l}
\mathit{factorial} = \mu\, x\,.\\
\quad \textbf{dmatch}\ \ x\\
\qquad \mathsf{0} \implies \mathbf{!}\mathsf{fact\text{-}1}\\
\qquad \mathsf{s}(y) \implies \textbf{dlet}\ \ \mathit{eq1} = \mathbf{!}\mathit{factorial}\ \ y\\
\qquad\qquad\qquad\qquad\quad \mathit{eq2} = \mathbf{!}\mathit{mult}\ \ x\ \ (\mathit{rhs}\ \ \mathit{eq1})\\
\qquad\qquad\qquad\ \ \textbf{in}\\
\qquad\qquad\qquad\quad \mathbf{!}\mathsf{fact\text{-}2}\ \ \mathit{eq1}\ \ \mathit{eq2}
\end{array}
$$

Lemma 9.9 *For all numerals x, the method call ($\mathbf{!}\mathit{factorial}\ \ x$) results in the equation $\mathsf{fact}(x) = y$, where the value of the numeral y is the factorial of the value of x.*

Next, we want to write an "evaluation method" *Eval* that takes an arbitrary expression e and derives the equation $e = x$, where x is the numeral representing the value of e. That is, *Eval* will essentially compute the mapping V, where numerals are identified with numbers.

$$
\begin{array}{l}
\mathit{Eval} = \mu\, e\,.\\
\quad \textbf{dmatch}\ e\\
\qquad \mathsf{0} \implies \mathbf{!}\mathsf{ref}\ \ \mathsf{0}\\
\qquad \mathsf{s}(e_1) \implies \mathbf{!}\mathsf{s\text{-}cong}\ \ (\mathbf{!}\mathit{Eval}\ \ e_1)\\
\qquad e_1 + e_2 \implies \textbf{dlet}\ \ \mathit{eq}_1 = \mathbf{!}\mathit{Eval}\ \ e_1\\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{eq}_2 = \mathbf{!}\mathit{Eval}\ \ e_2\\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{sum} = \mathbf{!}\mathit{add}\ \ (\mathit{rhs}\ \ \mathit{eq}_1)\ \ (\mathit{rhs}\ \ \mathit{eq}_2)\\
\qquad\qquad\qquad\quad\ \textbf{in}\\
\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{tran}\ \ (\mathbf{!}\mathsf{+\text{-}cong}\ \ \mathit{eq}_1\ \ \mathit{eq}_2)\ \ \mathit{sum}\\
\qquad e_1 * e_2 \implies \textbf{dlet}\ \ \mathit{eq}_1 = \mathbf{!}\mathit{Eval}\ \ e_1\\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{eq}_2 = \mathbf{!}\mathit{Eval}\ \ e_2\\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{product} = \mathbf{!}\mathit{mult}\ \ (\mathit{rhs}\ \ \mathit{eq}_1)\ \ (\mathit{rhs}\ \ \mathit{eq}_2)\\
\qquad\qquad\qquad\quad\ \textbf{in}\\
\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{tran}\ \ (\mathbf{!}\mathsf{*\text{-}cong}\ \ \mathit{eq}_1\ \ \mathit{eq}_2)\ \ \mathit{product}\\
\qquad \mathsf{fact}(e_1) \implies \textbf{dlet}\ \ \mathit{eq} = \mathbf{!}\mathit{Eval}\ \ e_1\\
\qquad\qquad\qquad\qquad\quad\ \ \mathit{fact} = \mathbf{!}\mathit{factorial}\ \ (\mathit{rhs}\ \ \mathit{eq})\\
\qquad\qquad\qquad\quad\ \textbf{in}\\
\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{tran}\ \ (\mathbf{!}\mathsf{fact\text{-}cong}\ \ \mathit{eq})\ \ \mathit{fact}
\end{array}
$$

As usual, the method proceeds by structural recursion on the input expression e. For the base case, when e is the numeral $\mathsf{0}$, we simply invoke the reflexivity method to derive the equality $\mathsf{0} = \mathsf{0}$. When e is of the form $\mathsf{s}(e_1)$, a recursive call $\mathbf{!}\mathit{Eval}\ \ e_1$

will return an equation of the form $e_1 = x_1$, where the numeral x_1 is the value of e_1. Therefore, the result that we want is $\mathsf{s}(e_1) = \mathsf{s}(x_1)$, which we can easily obtain by applying the method s-cong to the equation $e_1 = x_1$. When e is of the form $e_1 + e_2$, we recursively call $!Eval\ e_1$ and $!Eval\ e_2$, obtaining equations of the form $e_1 = x_1$ and $e_2 = x_2$, where the numerals x_1 and x_2 are the values of e_1 and e_2, respectively. Hence, the result that we want is

$$e_1 + e_2 = z \tag{9.2}$$

where z is the sum of x_1 and x_2. To obtain z, we use our numeral-addition method *add*: the call $!add\ x_1\ x_2$ then returns the equation

$$x_1 + x_2 = z. \tag{9.3}$$

Now by applying +-cong to the equations $e_1 = x_1$ and $e_2 = x_2$ we get $e_1 + e_2 = x_1 + x_2$; and applying transitivity to this and 9.3 we get the desired 9.2. The remaining cases are similar.

It is instructive to compare this method with an evaluation *algorithm*, expressed as a function. An algorithm would evaluate an expression such as $e_1 + e_2$ simply by evaluating the subexpressions e_1 and e_2 and adding the resulting numerals. This is more efficient but skips some steps which are important in understanding *why* we should accept the final numeral as being equal to the input expression. The method makes those steps explicit, and by doing so it not only produces the desired result but also proves that the result is correct.

Lemma 9.10 *For all expressions e, the method call* $(!Eval\ e)$ *results in a true equation of the form $e = x$.*

Finally, we note that the *Eval* method can be used to solve the validity problem for *Arith*. Suppose we are given a valid (true) equation of the form $e_1 = e_2$. We evaluate e_1, resulting in $e_1 = x_1$, then we evaluate e_2, resulting in $e_2 = x_2$. Now if $e_1 = e_2$ is indeed true then by the above lemma we must have $V[\![x_1]\!] = V[\![x_2]\!]$. But two numerals have identical values iff they are syntactically identical, hence the aforementioned equations must be of the form $e_1 = x$ and $e_2 = x$. By transposing the latter equation via symmetry and then using transitivity we obtain the desired $e_1 = e_2$. This is a simple manifestation of a general principle in equational logic, where two terms are shown equal by reducing both to a common normal form. Thus we arrive at the following method for recognizing all and only the valid sentences of *Arith*:

$prove = \mu\, eq\,.$
  dlet $eq_1 = !Eval\ (lhs\ eq)$
        $eq_2 = !Eval\ (rhs\ eq)$
  in
    !tran eq_1 (!sym eq_2)

It follows from our preceding discussion that the system is complete and decidable. We summarize:

Theorem 9.11 *The system Arith is consistent, sound, complete, decidable, and its validity problem is solvable.*

9.3 Recursive methods

We often present a method ψ in the form $\psi = \mu\,\overrightarrow{I}\,.\,D$, where ψ may occur in D. Although we have offered no way of desugaring something like this by itself into the $\lambda\mu$-calculus, we can always desugar any deduction D_0 that uses ψ into pure $\lambda\mu$-calculus as follows: **dletrec** $\psi = \mu\,\overrightarrow{I}\,.\,D$ **in** D_0. For instance, the method call

$$\mathbf{!}\,add\ \ \mathsf{s}(0)\ \ \mathsf{s}(0)$$

is desugared into

$$\textbf{dletrec}\ add = \mu\,x, y\,.\,D\ \textbf{in}\ \mathbf{!}\,add\ \ \mathsf{s}(0)\ \ \mathsf{s}(0). \tag{9.4}$$

(where D is the body of the add method, as shown in Section 9.2). To convince the reader that the semantics of the $\lambda\mu$-calculus "work", and that our various desugarings (especially those involving recursive methods) yield the right results, we will fully work out a derivation showing that evaluating the above deduction in the empty assumption base produces the sentence $\mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0))$.

First, suppose that the **dmatch** body of add has been desugared into

$$D = (\textbf{dcond}\ \ \textbf{app}(\mathsf{equal}, y, 0)\ \ \textbf{dapp}(\mathsf{plus\text{-}1}, x)\ \ \textbf{dapp}(\mathsf{plus\text{-}2}, \textbf{dapp}(add, x, \textbf{app}(pred, y))))$$

where $pred$ is a primitive unary function with the obvious semantics. Then 9.4 desugars into

$$D_{top} = \textbf{dapp}(\mu\,add\,.\,\textbf{dapp}(add, \mathsf{s}(0), \mathsf{s}(0)), \textbf{app}(Y, \lambda\,add\,.\,\mu\,x, y\,.\,D))$$

where, from Section 8.6.3,

$$Y = \lambda\,F\,.\,\textbf{app}(\lambda\,x\,.\,\textbf{app}(F, \textbf{app}(x, x)), \lambda\,x\,.\,\textbf{app}(F, \textbf{app}(x, x))).$$

We will show that $\emptyset \vdash D_{top} \leadsto \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0))$.

First, by axiom [R1] we have

$$\emptyset \vdash D_{top} \leadsto \textbf{dapp}(\textbf{app}(Y, E), \mathsf{s}(0), \mathsf{s}(0)) \tag{9.5}$$

where

$$E = \lambda\,add\,.\,\mu\,x, y\,.\,D. \tag{9.6}$$

Now letting

$$M = \lambda\, x \,.\, \mathbf{app}(E, \mathbf{app}(x, x)) \tag{9.7}$$

we get, again from [R1],

$$\emptyset \vdash \mathbf{app}(Y, E) \rightsquigarrow \mathbf{app}(M, M). \tag{9.8}$$

Next, from [R1],

$$\emptyset \vdash \mathbf{app}(M, M) \rightsquigarrow \mathbf{app}(E, \mathbf{app}(M, M)) \tag{9.9}$$

hence from 9.8, 9.9, and [R11] we get

$$\emptyset \vdash \mathbf{app}(Y, E) \rightsquigarrow \mathbf{app}(E, \mathbf{app}(M, M)). \tag{9.10}$$

Again from [R1], we have

$$\emptyset \vdash \mathbf{app}(E, \mathbf{app}(M, M)) \rightsquigarrow \mu\, x, y \,.\, D[\mathbf{app}(M, M)/add] \tag{9.11}$$

hence from 9.10, 9.11, and [R11] we get

$$\emptyset \vdash \mathbf{app}(Y, E) \rightsquigarrow \mu\, x, y \,.\, D[\mathbf{app}(M, M)/add]. \tag{9.12}$$

Accordingly, 9.5, 9.12, and [R4] yield

$$\emptyset \vdash D_{top} \rightsquigarrow \mathbf{dapp}(\mu\, x, y \,.\, D[\mathbf{app}(M, M)/add], \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0})) \tag{9.13}$$

From [R2] we have

$$\emptyset \vdash \mathbf{dapp}(\mu\, x, y \,.\, D[\mathbf{app}(M, M)/add], \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0})) \rightsquigarrow D' \tag{9.14}$$

where

$$D' = \begin{array}{l} (\mathbf{dcond}\quad \mathbf{app}(\mathsf{equal}, \mathsf{s}(\mathsf{0}), \mathsf{0}) \\ \quad\ \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})) \\ \quad\ \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{s}(\mathsf{0}))))) \end{array}.$$

But in accordance with the prescribed desugaring of **dcond**,

$$D' = \begin{array}{l} \mathbf{dapp}(\mathbf{app}(\mathsf{if}_D, \mathbf{app}(\mathsf{equal}, \mathsf{s}(\mathsf{0}), \mathsf{0})), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{s}(\mathsf{0}))))) \end{array} \tag{9.15}$$

hence from 9.13, 9.14, 9.15, and [R11] we get

$$\emptyset \vdash D_{top} \rightsquigarrow \begin{array}{l}\mathbf{dapp}(\mathbf{app}(\mathsf{if}_D, \mathbf{app}(\mathsf{equal}, \mathsf{s}(0), 0)),\\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(0)),\\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(0), \mathbf{app}(\mathit{pred}, \mathsf{s}(0))))).\end{array} \tag{9.16}$$

Since

$$\emptyset \vdash \mathbf{app}(\mathsf{equal}, \mathsf{s}(0), 0) \rightsquigarrow \mathsf{false}$$

the δ-rules for if_D give

$$\emptyset \vdash \mathbf{app}(\mathsf{if}_D, \mathbf{app}(\mathsf{equal}, \mathsf{s}(0), 0)) \rightsquigarrow \mu\, x, y \,.\, \mathbf{dapp}(\mathsf{claim}, y) \tag{9.17}$$

so 9.16, 9.17, and [R4] yield

$$\emptyset \vdash D_{top} \rightsquigarrow \begin{array}{l}\mathbf{dapp}(\mu\, x, y \,.\, \mathbf{dapp}(\mathsf{claim}, y),\\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(0)),\\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(0), \mathbf{app}(\mathit{pred}, \mathsf{s}(0))))).\end{array} \tag{9.18}$$

and from [R2] and [R11] we get

$$\emptyset \vdash D_{top} \rightsquigarrow \mathbf{dapp}(\mathsf{claim}, \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(0), \mathbf{app}(\mathit{pred}, \mathsf{s}(0))))). \tag{9.19}$$

Now $\emptyset \vdash \mathbf{app}(\mathit{pred}, \mathsf{s}(0)) \rightsquigarrow 0$, hence 9.19 via [R4] gives

$$\emptyset \vdash D_{top} \rightsquigarrow \mathbf{dapp}(\mathsf{claim}, \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(0), 0))). \tag{9.20}$$

Next, let

$$K = \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(0), 0) \tag{9.21}$$

From 9.9, 9.11, and [R11] we conclude

$$\emptyset \vdash \mathbf{app}(M, M) \rightsquigarrow \mu\, x, y \,.\, D[\mathbf{app}(M, M)/\mathit{add}]$$

thus from 9.21 and [R4] we get

$$\emptyset \vdash K \rightsquigarrow \mathbf{dapp}(\mu\, x, y \,.\, D[\mathbf{app}(M, M)/\mathit{add}], \mathsf{s}(0), 0) \tag{9.22}$$

Now [R1] gives

$$\emptyset \vdash \mathbf{dapp}(\mu\, x, y \,.\, D[\mathbf{app}(M, M)/\mathit{add}], \mathsf{s}(0), 0) \rightsquigarrow D[\mathbf{app}(M, M), \mathsf{s}(0), 0/\mathit{add}, x, y] \tag{9.23}$$

so 9.22, 9.23, and [R11] give

$$\emptyset \vdash K \rightsquigarrow D[\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathsf{0}/add, x, y]$$

and since

$$D[\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathsf{0}/add, x, y] = \begin{array}{l}(\mathbf{dcond} \quad \mathbf{app}(\mathsf{equal}, \mathsf{0}, \mathsf{0}) \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})) \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{0}))))\end{array}$$

we get

$$\emptyset \vdash K \rightsquigarrow \begin{array}{l}(\mathbf{dcond} \quad \mathbf{app}(\mathsf{equal}, \mathsf{0}, \mathsf{0}) \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})) \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{0}))))\end{array} \tag{9.24}$$

or, according to the desugaring of **dcond**,

$$\emptyset \vdash K \rightsquigarrow \begin{array}{l}\mathbf{dapp}(\mathbf{app}(\mathsf{if}_D, \mathbf{app}(\mathsf{equal}, \mathsf{0}, \mathsf{0})), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{0})))).\end{array} \tag{9.25}$$

Further, $\emptyset \vdash \mathbf{app}(\mathsf{equal}, \mathsf{0}, \mathsf{0}) \rightsquigarrow \mathsf{true}$, hence

$$\emptyset \vdash \mathbf{app}(\mathsf{if}_D, \mathbf{app}(\mathsf{equal}, \mathsf{0}, \mathsf{0})) \rightsquigarrow \mu\, x, y \,.\, \mathbf{dapp}(\mathsf{claim}, x)$$

and from 9.25 and [R4] we get

$$\emptyset \vdash K \rightsquigarrow \begin{array}{l}\mathbf{dapp}(\mu\, x, y \,.\, \mathbf{dapp}(\mathsf{claim}, x), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0})), \\ \quad \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathbf{dapp}(\mathbf{app}(M, M), \mathsf{s}(\mathsf{0}), \mathbf{app}(pred, \mathsf{0}))))\end{array} \tag{9.26}$$

so, from [R2] and [R11],

$$\emptyset \vdash K \rightsquigarrow \mathbf{dapp}(\mathsf{claim}, \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(\mathsf{0}))). \tag{9.27}$$

Now, by the δ-rules of plus-1,

$$\emptyset \vdash \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(0)) \rightsquigarrow \mathsf{s}(0) + 0 = \mathsf{s}(0) \tag{9.28}$$

while

$$\{\mathsf{s}(0) + 0 = \mathsf{s}(0)\} \vdash \mathbf{dapp}(\mathsf{claim}, \mathsf{s}(0) + 0 = \mathsf{s}(0)) \rightsquigarrow \mathsf{s}(0) + 0 = \mathsf{s}(0) \tag{9.29}$$

hence 9.28, 9.29, and [R7] give

$$\emptyset \vdash \mathbf{dapp}(\mathsf{claim}, \mathbf{dapp}(\mathsf{plus\text{-}1}, \mathsf{s}(0))) \rightsquigarrow \mathsf{s}(0) + 0 = \mathsf{s}(0). \tag{9.30}$$

Therefore, from 9.27, 9.30, and [R11] we get

$$\emptyset \vdash K \rightsquigarrow \mathsf{s}(0) + 0 = \mathsf{s}(0). \tag{9.31}$$

By the δ-rules of plus-2,

$$\{\mathsf{s}(0) + 0 = \mathsf{s}(0)\} \vdash \mathbf{dapp}(\mathsf{plus\text{-}2}, \mathsf{s}(0) + 0 = \mathsf{s}(0)) \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0)) \tag{9.32}$$

so now from 9.31, 9.32, and [R7] we get

$$\emptyset \vdash \mathbf{dapp}(\mathsf{plus\text{-}2}, K) \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0)). \tag{9.33}$$

Moreover,

$$\{\mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0))\} \vdash \mathbf{dapp}(\mathsf{claim}, \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0))) \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0)) \tag{9.34}$$

hence from 9.33, 9.34, and [R7] we get

$$\vdash \mathbf{dapp}(\mathsf{claim}, \mathbf{dapp}(\mathsf{plus\text{-}2}, K)) \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0)). \tag{9.35}$$

Finally, from 9.20, 9.35, and [R11] we obtain the desired

$$\emptyset \vdash D_{top} \rightsquigarrow \mathsf{s}(0) + \mathsf{s}(0) = \mathsf{s}(\mathsf{s}(0)).$$

9.4 Natural deduction in the $\lambda\mu$-calculus, propositional case

In this section we introduce a $\lambda\mu$ system for classic zero-order natural deduction, $\lambda\mu$–$\mathcal{CND}_0$. We show that $\lambda\mu$–$\mathcal{CND}_0$ subsumes propositional $\mathcal{CND}$, and goes further by providing the naming and abstraction mechanisms missing from the latter (see Section 4.9). We show that $\lambda\mu$–$\mathcal{CND}_0$ is sound and complete (completeness follows readily from the inclusion of $\mathcal{CND}$), present several examples of deductions written in it, and show how the deductive abstraction mechanisms of the $\lambda\mu$-calculus allow us to formulate powerful "derived inference rules" as methods.

The constants of $\lambda\mu$–$\mathcal{CND}_0$ are the following:

- The propositions generated by the abstract grammar

$$P \ ::= \mathsf{true} \mid \mathsf{false} \mid A_i \mid \neg P \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P \Leftrightarrow Q$$

where $A_1, A_2, \ldots$ are distinct propositional atoms. These will be the *sentences* of the system. Accordingly, assumption bases will be sets of propositions.

- The primitive functions not, and, or, if, and iff. These will be used for building propositions. Their semantics are given by the following δ-rule schemas:

$$\begin{gathered}
\beta \vdash \mathbf{app}(\mathsf{not}, P) \rightsquigarrow \neg P \\
\beta \vdash \mathbf{app}(\mathsf{and}, P, Q) \rightsquigarrow P \wedge Q \\
\beta \vdash \mathbf{app}(\mathsf{or}, P, Q) \rightsquigarrow P \vee Q \\
\beta \vdash \mathbf{app}(\mathsf{if}, P, Q) \rightsquigarrow P \Rightarrow Q \\
\beta \vdash \mathbf{app}(\mathsf{iff}, P, Q) \rightsquigarrow P \Leftrightarrow Q
\end{gathered}$$

Note that all five of these functions are context-independent, and satisfy requirements **PF1** and **PF2**.

- The primitive methods T-axiom, F-axiom, mp, mt, dn, both, left-and, right-and, cd, left-either, right-either, equiv, left-iff, right-iff, and absurd. Their semantics are specified by the δ-evaluation axioms in Figure 9.2. The reader will verify that the axioms are such that all fifteen methods satisfy **PM1** and **PM2**.

In addition, $\lambda\mu$–$\mathcal{CND}_0$ has one special deductive form: **assume**(M, D), which we will write as **assume** M **in** D. Its semantics are given by the rule

$$\frac{\beta \vdash M \rightsquigarrow^* P \qquad \beta \cup \{P\} \vdash D \rightsquigarrow Q}{\beta \vdash \mathbf{assume}(M, D) \rightsquigarrow P \Rightarrow Q} \tag{9.36}$$

which, as the reader will verify, satisfies the requirements of Section 8.2.3.

We introduce the form **assume** $I = M$ **in** D as syntax sugar for

dlet $I = M$
in
  assume I **in** D

Moreover, we introduce deductions of the form **suppose-absurd** M **in** D as abbreviations for

dlet $I =$ **assume** M **in** D
in
  dapp(mt, I, **dapp**(F-axiom))

$$\emptyset \vdash \mathbf{dapp}(\textsf{T-axiom}) \leadsto \textsf{true}$$
$$\emptyset \vdash \mathbf{dapp}(\textsf{F-axiom}) \leadsto \neg\textsf{false}$$
$$\{P \Rightarrow Q, P\} \vdash \mathbf{dapp}(\textsf{mp}, P \Rightarrow Q, P) \leadsto Q$$
$$\{P \Rightarrow Q, \neg Q\} \vdash \mathbf{dapp}(\textsf{mt}, P \Rightarrow Q, \neg Q) \leadsto \neg P$$
$$\{\neg\neg P\} \vdash \mathbf{dapp}(\textsf{dn}, \neg\neg P) \leadsto P$$
$$\{P, Q\} \vdash \mathbf{dapp}(\textsf{both}, P, Q) \leadsto P \wedge Q$$
$$\{P \wedge Q\} \vdash \mathbf{dapp}(\textsf{left-and}, P \wedge Q) \leadsto P$$
$$\{P \wedge Q\} \vdash \mathbf{dapp}(\textsf{right-and}, P \wedge Q) \leadsto Q$$
$$\{P\} \vdash \mathbf{dapp}(\textsf{left-either}, P, Q) \leadsto P \vee Q$$
$$\{Q\} \vdash \mathbf{dapp}(\textsf{right-either}, P, Q) \leadsto P \vee Q$$
$$\{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\} \vdash \mathbf{dapp}(\textsf{cd}, P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q) \leadsto Q$$
$$\{P \Rightarrow Q, Q \Rightarrow P\} \vdash \mathbf{dapp}(\textsf{equiv}, P \Rightarrow Q, Q \Rightarrow P) \leadsto P \Leftrightarrow Q$$
$$\{P \Leftrightarrow Q\} \vdash \mathbf{dapp}(\textsf{left-iff}, P \Leftrightarrow Q) \leadsto P \Rightarrow Q$$
$$\{P \Leftrightarrow Q\} \vdash \mathbf{dapp}(\textsf{right-iff}, P \Leftrightarrow Q) \leadsto Q \Rightarrow P$$
$$\{P, \neg P\} \vdash \mathbf{dapp}(\textsf{absurd}, P, \neg P) \leadsto \textsf{false}$$

Figure 9.2: δ-evaluation axioms for the primitive methods of $\lambda\mu$–$\mathcal{CND}_0$.

for I that does not occur in M or in D. We show the correctness of this translation by proving that it captures the intended semantics:

Lemma 9.12 *If* $\beta \vdash M \leadsto^* P$ *and* $\beta \cup \{P\} \vdash D \leadsto \textsf{false}$ *then*

$$\beta \vdash \textbf{suppose-absurd } M \textbf{ in } D \leadsto \neg P.$$

Proof: By definition, **suppose-absurd** M **in** D abbreviates **dlet** $I = D_1$ **in** D_2, where $D_1 = \textbf{assume } M \textbf{ in } D$ and $D_2 = \mathbf{dapp}(\textsf{mt}, I, \mathbf{dapp}(\textsf{F-axiom}))$. Therefore, by Lemma 8.9, the desired conclusion will follow if we can only show:

(a) $\beta \vdash D_1 \leadsto P \Rightarrow \textsf{false}$; and

(b) $\beta \cup \{P \Rightarrow \textsf{false}\} \vdash D_2[P \Rightarrow \textsf{false}/I] \leadsto \neg P$, i.e.,

$$\beta \cup \{P \Rightarrow \textsf{false}\} \vdash \mathbf{dapp}(\textsf{mt}, P \Rightarrow \textsf{false}, \mathbf{dapp}(\textsf{F-axiom})) \leadsto \neg P.$$

Now (a) holds by the suppositions $\beta \vdash M \leadsto^* P$, $\beta \cup \{P\} \vdash D \leadsto \textsf{false}$, and the semantics of **assume**; while (b) holds owing to [R7] and the semantics of $\textsf{mt}$ and $\textsf{F-axiom}$. ■

Let us work out a detailed example showing a derivation of a judgment. We will derive the judgment $\{Q\} \vdash D \rightsquigarrow (P \Rightarrow P) \wedge Q$, where D is the deduction

$$\begin{array}{l} \mathsf{!both}\ (\textbf{assume}\ P\ \textbf{in}\ \mathsf{!claim}\ P) \\ \qquad ((\lambda\, x\, .\, x)\ \ Q) \end{array}$$

1.	$\{P\} \vdash \mathsf{!claim}\ \ P \rightsquigarrow P$	[R12]
2.	$\{P, Q\} \vdash \mathsf{!claim}\ \ P \rightsquigarrow P$	1, [R10]
3.	$\{Q\} \vdash \textbf{assume}\ P\ \textbf{in}\ \mathsf{!claim}\ \ P \rightsquigarrow P \Rightarrow P$	2, 9.36
4.	$\{P \Rightarrow P, Q\} \vdash ((\lambda\, x\, .\, x)\ \ Q) \rightsquigarrow Q$	[R1]
5.	$\{P \Rightarrow P, Q\} \vdash \mathsf{!both}\ \ P \Rightarrow P\ \ ((\lambda\, x\, .\, x)\ \ Q) \rightsquigarrow \mathsf{!both}\ \ P \Rightarrow P\ \ Q$	4, [R6]
6.	$\{P \Rightarrow P, Q\} \vdash \mathsf{!both}\ \ P \Rightarrow P\ \ Q \rightsquigarrow (P \Rightarrow P) \wedge Q$	δ-axiom
7.	$\{P \Rightarrow P, Q\} \vdash \mathsf{!both}\ \ P \Rightarrow P\ \ ((\lambda\, x\, .\, x)\ \ Q) \rightsquigarrow (P \Rightarrow P) \wedge Q$	5, 6, [R11]
8.	$\{Q\} \vdash \mathsf{!both}\ (\textbf{assume}\ P\ \textbf{in}\ \mathsf{!claim}\ \ P)\ \ ((\lambda\, x\, .\, x)\ \ Q) \rightsquigarrow (P \Rightarrow P) \wedge Q$	3, 7, [R7]

We present a second, slightly more involved example. Let D be the deduction

$$\begin{array}{l} \mathsf{!}((\lambda\, x\, .\, x)\ \mathsf{both})\ \ (\textbf{assume}\ R\ \ \textbf{in}\ \ \mathsf{!claim}\ \ R) \\ \qquad (\mathsf{!right\text{-}either}\ \ \neg R\ \ \ (\textbf{assume}\ \ P \wedge Q\ \ \textbf{in}\ \ \mathsf{!both}\ \ (\mathsf{!right\text{-}and}\ \ P \wedge Q) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{!left\text{-}and}\ \ P \wedge Q))). \end{array}$$

The derivation shown below establishes the judgment

$$\emptyset \vdash D \rightsquigarrow R \Rightarrow R \wedge [\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)].$$

Note the crucial role played by [R7].

1. $\{P \wedge Q\} \vdash \mathsf{!left\text{-}and}\ \ P \wedge Q \rightsquigarrow P$ δ-axiom
2. $\{P \wedge Q\} \vdash \mathsf{!right\text{-}and}\ \ P \wedge Q \rightsquigarrow Q$ δ-axiom
3. $\{P \wedge Q, P, Q\} \vdash \mathsf{!both}\ \ Q\ \ P \rightsquigarrow Q \wedge P$ δ-axiom
4. $\{P \wedge Q, Q\} \vdash \mathsf{!both}\ \ Q\ \ (\mathsf{!left\text{-}and}\ \ P \wedge Q) \rightsquigarrow Q \wedge P$ 1, 3, [R7]
5. $\{P \wedge Q\} \vdash \mathsf{!both}\ \ (\mathsf{!right\text{-}and}\ \ P \wedge Q)\ \ (\mathsf{!left\text{-}and}\ \ P \wedge Q) \rightsquigarrow Q \wedge P$ 2, 4, [R7]
6. $\emptyset \vdash \textbf{assume}\ P \wedge Q\ \textbf{in}\ \mathsf{!both}\ \ (\mathsf{!right\text{-}and}\ \ P \wedge Q)\ \ (\mathsf{!left\text{-}and}\ \ P \wedge Q) \rightsquigarrow$
   $P \wedge Q \Rightarrow Q \wedge P$ 5, 9.36
7. $\{P \wedge Q \Rightarrow Q \wedge P\} \vdash \mathsf{!right\text{-}either}\ \ \neg R\ \ (P \wedge Q \Rightarrow Q \wedge P) \rightsquigarrow$
   $\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)$ δ-axiom
8. $\emptyset \vdash \mathsf{!right\text{-}either}\ \ \neg R$
   $\textbf{assume}\ P \wedge Q\ \textbf{in}\ \mathsf{!both}\ \ (\mathsf{!right\text{-}and}\ \ P \wedge Q)$
   $(\mathsf{!left\text{-}and}\ \ P \wedge Q) \rightsquigarrow$
   $\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)$ 6, 7, [R7]

9. $\{R\} \vdash \,!\mathsf{claim}\ \ R \rightsquigarrow R \quad \delta$-axiom

10. $\emptyset \vdash \textbf{assume}\ R\ \textbf{in}\ !\mathsf{claim}\ \ R \rightsquigarrow R \Rightarrow R \quad 9, 9.36$

11. $\emptyset \vdash (\lambda\, x\, .\, x)\ \ \mathsf{both} \rightsquigarrow \mathsf{both} \quad$ [R1]

12. $$\begin{aligned}\emptyset \vdash D \rightsquigarrow\, !\mathsf{both}\ \ &(\textbf{assume}\ R\ \textbf{in}\ !\mathsf{claim}\ \ R)\\ &(!\mathsf{right\text{-}either}\ \ \neg R\\ &\qquad(\textbf{assume}\ P \wedge Q\ \textbf{in}\ !\mathsf{both}\ \ (!\mathsf{right\text{-}and}\ \ P \wedge Q)\\ &\qquad\qquad(!\mathsf{left\text{-}and}\ \ P \wedge Q)))\quad 11, [R4]\end{aligned}$$

13. $$\begin{aligned}&\{R \Rightarrow R, \neg R \vee (P \wedge Q \Rightarrow Q \wedge P)\} \vdash \,!\mathsf{both}\ \ R \Rightarrow R\ \ \neg R \vee (P \wedge Q \Rightarrow Q \wedge P) \rightsquigarrow\\ &\qquad R \Rightarrow R \wedge \neg R \vee (P \wedge Q \Rightarrow Q \wedge P) \quad \delta\text{-axiom}\end{aligned}$$

14. $$\begin{aligned}\{R \Rightarrow R\} \vdash \,!\mathsf{both}\ \ &R \Rightarrow R\\ &(!\mathsf{right\text{-}either}\ \ \neg R\\ &\qquad(\textbf{assume}\ P \wedge Q\ \textbf{in}\ !\mathsf{both}\ \ (!\mathsf{right\text{-}and}\ \ P \wedge Q)\\ &\qquad\qquad(!\mathsf{left\text{-}and}\ \ P \wedge Q))) \rightsquigarrow\\ &R \Rightarrow R \wedge [\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)] \quad 8, 13, [R7]\end{aligned}$$

15. $$\begin{aligned}\emptyset \vdash \,!\mathsf{both}\ \ &(\textbf{assume}\ R\ \textbf{in}\ !\mathsf{claim}\ \ R)\\ &(!\mathsf{right\text{-}either}\ \ \neg R\\ &\qquad(\textbf{assume}\ P \wedge Q\ \textbf{in}\ !\mathsf{both}\ \ (!\mathsf{right\text{-}and}\ \ P \wedge Q)\\ &\qquad\qquad(!\mathsf{left\text{-}and}\ \ P \wedge Q))) \rightsquigarrow\\ &R \Rightarrow R \wedge [\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)] \quad 10, 14, [R7]\end{aligned}$$

16. $\emptyset \vdash D \rightsquigarrow R \Rightarrow R \wedge [\neg R \vee (P \wedge Q \Rightarrow Q \wedge P)] \quad 12, 15,$ [R11]

We will now show that $\lambda\mu$–$\mathcal{CND}_0$ subsumes propositional $\mathcal{CND}$, in the sense that every $\mathcal{CND}$ deduction can be straighforwardly desugared into a $\lambda\mu$–$\mathcal{CND}_0$ deduction. Hypothetical deductions of the form **assume** P **in** D correspond directly to **assume** deductions in $\lambda\mu$–$\mathcal{CND}_0$; while proof compositions $D_1; D_2$ correspond to **dlet**s, and thus essentially to nested method applications.

Specifically, we define a function $[\![\cdot]\!]$ from $\mathcal{CND}$ deductions to $\lambda\mu$–$\mathcal{CND}_0$ deductions as follows:

$$\begin{aligned}[\![\textbf{true}]\!] &= \textbf{dapp}(\mathsf{T\text{-}axiom})\\ [\![\neg\textbf{false}]\!] &= \textbf{dapp}(\mathsf{F\text{-}axiom})\\ [\![P]\!] &= \textbf{dapp}(\mathsf{claim}, P)\ \ (\text{for } P \notin \{\textbf{true}, \neg\textbf{false}\})\\ [\![\mathit{Rule}\ P_1, \ldots, P_n]\!] &= \textbf{dapp}(\mathit{Rule}^*, P_1, \ldots, P_n)\\ [\![\textbf{assume}\ P\ \textbf{in}\ D]\!] &= \textbf{assume}\ P\ \textbf{in}\ [\![D]\!]\\ [\![D_1; D_2]\!] &= \textbf{dlet}\ I = [\![D_1]\!]\ \textbf{in}\ [\![D_2]\!]\end{aligned}$$

for fresh I (meaning that every time the last clause is applied, a different I should be used[1]). Also, for every primitive rule *Rule* of $\mathcal{CND}$ (listed in Figure 4.1), *Rule** denotes the corresponding primitive method of $\lambda\mu$–$\mathcal{CND}_0$.[2]

The next result now follows by a straightforward induction on D.

Theorem 9.13 $\beta \vdash_{\mathcal{CND}} D \rightsquigarrow P$ *iff* $\beta \vdash_{\lambda\mu\text{–}\mathcal{CND}_0} [\![D]\!] \rightsquigarrow P$.

Proof: By structural induction on D. When D is a simple claim the result follows by virtue of T-axiom, F-axiom, and the $\lambda\mu$ rules [R12] and [R10]. When D is of the form *Rule* $P_1, \ldots, P_n$, the result follows from a case analysis of *Rule*, in accordance with the rules shown in Figure 4.3 and Figure 9.2. When D is of the form **assume** P_1 **in** D_1 then P must be of the form $P_1 \Rightarrow P_2$ and

$\beta \vdash_{\mathcal{CND}} D \rightsquigarrow P_1 \Rightarrow P_2$	iff	
$\beta \cup \{P_1\} \vdash_{\mathcal{CND}} D_1 \rightsquigarrow P_2$	iff	(from the inductive hypothesis)
$\beta \cup \{P_1\} \vdash_{\lambda\mu\text{–}\mathcal{CND}_0} [\![D_1]\!] \rightsquigarrow P_2$	iff	(from 9.36)
$\beta \vdash_{\lambda\mu\text{–}\mathcal{CND}_0}$ **assume** P_1 **in** $[\![D_1]\!] \rightsquigarrow P_1 \Rightarrow P_2$.		

Finally, when D is of the form $D_1; D_2$ we have

$\beta \vdash_{\mathcal{CND}} D_1; D_2 \rightsquigarrow P$	iff	
$\beta \vdash_{\mathcal{CND}} D_1 \rightsquigarrow P_1$ and $\beta \cup \{P_1\} \vdash_{\mathcal{CND}} D_2 \rightsquigarrow P$	iff	(inductively)
$\beta \vdash_{\lambda\mu\text{–}\mathcal{CND}_0} [\![D_1]\!] \rightsquigarrow P_1$ and $\beta \cup \{P_1\} \vdash_{\lambda\mu\text{–}\mathcal{CND}_0} [\![D_2]\!] \rightsquigarrow P$	iff	(Corollary 8.10)
$\beta \vdash_{\lambda\mu\text{–}\mathcal{CND}_0}$ **dlet** $I = [\![D_1]\!]$ **in** $[\![D_2]\!] \rightsquigarrow P$.		

The result now follows by structural induction. ∎

Next we turn to soundness and completeness. We define logical entailment $\beta \models P$ as usual (see Section 4.6). Then, by Theorem 8.7, we can show $\lambda\mu$–$\mathcal{CND}_0$ to be sound provided we can show that $\models$ includes every primitive method and that the form **assume** preserves $\models$. A cursory inspection of the δ-evaluation axioms in Figure 9.2 and the rule 9.36 will suffice for both tasks. We conclude:

Theorem 9.14 (Soundness) $\lambda\mu$–$\mathcal{CND}_0$ *is sound, i.e.,* $\beta \models P$ *whenever* $\beta \vdash D \rightsquigarrow P$.

The completeness of $\lambda\mu$–$\mathcal{CND}_0$ follows from the completeness of $\mathcal{CND}$ and Theorem 9.13:

[1]In fact this is not essential (the same I could be used repeatedly), but it simplifies the proof of the correctness of the desugaring

[2]The correspondence being obvious: mt corresponds to **modus-tollens**, etc.

Theorem 9.15 (Completeness) *$\lambda\mu$–$\mathcal{CND}_0$ is complete. That is, if $\beta \models P$ then there is a deduction D such that $\beta \vdash D \leadsto P$.*

We now present sample $\lambda\mu$–$\mathcal{CND}_0$ deductions of some of the tautologies that were proved earlier in $\mathcal{CND}$ (Section 4.4). Comparing the two sets of deductions, we see that they are very similar (modulo the replacement of the composition operator ; by **dlet**, as explained in the proof of Theorem 9.13), the most noticeable difference being that $\lambda\mu$–$\mathcal{CND}_0$ deductions can be somewhat more compact than their $\mathcal{CND}$ counterparts due to their use of nested method applications. More essential differences will surface when we come to formulate methods.

$$\boxed{P \Rightarrow \neg\neg P}$$

Proof:

assume P **in**
  suppose-absurd $\neg P$ **in**
    !absurd P $\neg P$

$$\boxed{P \wedge Q \Rightarrow Q \wedge P}$$

Proof:

assume $x = P \wedge Q$ **in**
  !both (!right-and x)
        (!left-and x)

An alternative proof of the above tautology is:

assume $P \wedge Q$ **in**
  dlet *right* = !right-and $P \wedge Q$
       left = !left-and $P \wedge Q$
  in
    !both *right* *left*

$$\boxed{(P \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow Q \Rightarrow R)}$$

Proof:

assume $imp = P \wedge Q \Rightarrow R$ **in**
  assume P **in**
    assume Q **in**
      !mp imp (!both P Q)

$$\boxed{\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)}$$

Proof:

assume $\neg(P \vee Q)$ **in**
  dlet $not\text{-}P$ = **suppose-absurd** P **in**
                 !absurd (!left-either P Q) $\neg(P \vee Q)$
       $not\text{-}Q$ = **suppose-absurd** Q **in**
                 !absurd (!right-either P Q) $\neg(P \vee Q)$
  in
    !both $not\text{-}P$ $not\text{-}Q$

Finally, we will show how easy it is in $\lambda\mu$–$\mathcal{CND}_0$ to abstract away from concrete deductions towards general methods. Consider, for instance, the deduction schema shown above for establishing $P \Rightarrow \neg\neg P$. Abstracting over P results in the following method:

$m_1 = \mu\, P\, .$
 assume P **in**
   suppose-absurd $\neg P$ **in**
     !absurd P $\neg P$

This method can be applied to different propositions, producing different instances of the tautology schema $P \Rightarrow \neg\neg P$. For instance, the reader will verify that:

$$\emptyset \vdash !m_1\ \ A_2 \wedge A_5 \rightsquigarrow (A_2 \wedge A_5) \Rightarrow \neg\neg(A_2 \wedge A_5).$$

Note that m_1 does not require any premises, as the input P is not strictly used in the body of the method. In fact the body of m_1 does not have any free assumptions at all. It is for this reason that the application of m_1 always results in a tautology. We say that methods of this form (that do not require any premises and hence always result in tautologies) are *hypothetical*. Non-conservative methods will be called *strict*. Hypothetical methods can be viewed as derived axioms (or, more precisely, as axiom schemas), since they do not have any premises, whereas strict methods can be seen as derived rules that require one or more premises in order to yield a conclusion.

Hypothetical and strict methods are essentially equivalent, because any inference rule of the form

$$\frac{P_1, \ldots, P_n}{P}$$

is tantamount to the axiom

$$\frac{}{P_1 \Rightarrow \cdots \Rightarrow P_n \Rightarrow P}$$

Thus from any strict method with body D and $n > 0$ free assumptions $P_1, \ldots, P_n$ we may obtain a hypothetical method by replacing D with

$$\textbf{assume } P_1 \textbf{ in } \cdots \textbf{assume } P_n \textbf{ in } D.$$

And conversely, from any hypothetical method we may obtain a strict one by removing certain **assume**s. For instance, m_1 can be expressed in strict style as:

$m'_1 = \mu P$.
suppose-absurd $\neg P$ **in**
  !absurd P $\neg P$

In contrast to m_1, m'_1 requires the argument P to be a premise, i.e., to be in the assumption base at the time of invocation. If it is not, an error will occur. Thus m'_1 "takes P and derives $\neg\neg P$", and in that sense it is a more customary inference rule (if by an inference rule we mean one that takes certain propositions $P_1, \ldots, P_n$ *that have already been established* and produces, on that basis, some conclusion P).

Note that most primitive methods, such as modus ponens or double negation, are strict. Double negation, for example, requires its argument $\neg\neg P$ to hold (i.e., to be in the assumption base) in order to generate the conclusion P. Alternatively, we could have made dn hypothetical by having it produce, for a given P, the tautology $\neg\neg P \Rightarrow P$. But then every time we had $\neg\neg P$ and wanted to infer P from it we would have to first call dn in order to obtain the tautology $\neg\neg P \Rightarrow P$, and then apply modus ponens to this and $\neg\neg P$ in order to detach the desired P. This is somewhat roundabout, so we prefer to obtain P directly by applying dn to $\neg\neg P$. Accordingly, in what follows we will write most of our methods in strict style.

We continue with four methods dm_1, $dm_1{}'$, dm_2, and $dm_2{}'$, that can be seen as "derived inference rules" for De Morgan's laws:

$$\frac{\neg(P \vee Q)}{\neg P \wedge \neg Q}\ [dm_1] \qquad \frac{\neg P \wedge \neg Q}{\neg(P \vee Q)}\ [dm_1{}'] \qquad \frac{\neg(P \wedge Q)}{\neg P \vee \neg Q}\ [dm_2] \qquad \frac{\neg P \vee \neg Q}{\neg(P \wedge Q)}\ [dm_2{}']$$

Accordingly, the methods dm_1 and $dm_1{}'$ will be inverses, in the sense that the identities

$$!\,dm_1\ \ (!\,dm_1{}'\ \ P) = P$$

and

$$!\,dm_1{}'\ \ (!\,dm_1\ \ P) = P$$

will hold in any assumption base that contains P (with P of the right form; see the remark following the definition of dm_1); and likewise for dm_2 and $dm_2{}'$. We implement the four methods as follows:

$dm_1 = \mu\, premise\,.$
 dmatch *premise*
  $\neg(P \vee Q) \Longrightarrow$ **dlet** *not-P* = **suppose-absurd** P **in**
          $!\mathsf{absurd}\ \ (!\mathsf{left\text{-}either}\ \ P\ \ Q)\ \ \neg(P \vee Q)$
        not-Q = **suppose-absurd** Q **in**
          $!\mathsf{absurd}\ \ (!\mathsf{right\text{-}either}\ \ P\ \ Q)\ \ \neg(P \vee Q)$
      in
       $!\mathsf{both}$ *not-P* *not-Q*

Note that, by the semantics of **dmatch**, if the input *premise* is not of the form $\neg(P \vee Q)$ then no match will be found and an error will occur. This is quite appropriate, as we only intend the method to work for inputs of the aforementioned form. We continue with the remaining methods:

$dm_1{}' = \mu\, premise\,.$
 dmatch *premise*
  $\neg P \wedge \neg Q \Longrightarrow$ **dlet** $cond_1 = P \Rightarrow \mathsf{false}$ **by** **assume** P **in**
            $!\mathsf{absurd}\ \ P\ \ (!\mathsf{left\text{-}and}\ \ premise)$
        $cond_1 = Q \Rightarrow \mathsf{false}$ **by** **assume** Q **in**
            $!\mathsf{absurd}\ \ Q\ \ (!\mathsf{right\text{-}and}\ \ premise)$
      in
       $\neg(P \vee Q)$ **by** **suppose-absurd** $P \vee Q$ **in**
          $!\mathsf{cd}\ \ P \vee Q\ \ cond_1\ \ cond_2$

Next is dm_2:

$dm_2 = \mu\, premise\,.$
 dmatch *premise*

$$
\begin{array}{l}
\neg(P \wedge Q) \Longrightarrow \textbf{dlet}\ S = \neg\neg(\neg P \vee \neg Q)\ \textbf{by} \\
\qquad\qquad\qquad\qquad \textbf{suppose-absurd}\ H = \neg(\neg P \vee \neg Q)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad \textbf{dlet}\ S_1 = \neg\neg P \wedge \neg\neg Q\ \textbf{by}\ \mathbf{!}\,dm_1\ \ H \\
\qquad\qquad\qquad\qquad\qquad\quad S_2 = P\ \textbf{by}\ \mathbf{!}\mathsf{dn}\ \ (\mathbf{!}\mathsf{left\text{-}and}\ \ S_1) \\
\qquad\qquad\qquad\qquad\qquad\quad S_3 = Q\ \textbf{by}\ \mathbf{!}\mathsf{dn}\ \ (\mathbf{!}\mathsf{right\text{-}and}\ \ S_1) \\
\qquad\qquad\qquad\qquad\qquad\quad S_4 = P \wedge Q\ \textbf{by}\ \mathbf{!}\mathsf{both}\ \ S_2\ \ S_3 \\
\qquad\qquad\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{absurd}\ \ S_4\ \ \mathit{premise} \\
\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad \neg P \vee \neg Q\ \textbf{by}\ \mathbf{!}\mathsf{dn}\ \ S
\end{array}
$$

Here we have already begun to witness the benefits of the abstraction mechanism of methods: proposition S_1 above is derived by applying the previously defined method dm_1 to the hypothesis H. We conclude with $dm_2{}'$:

$$
\begin{array}{l}
dm_2{}' = \mu\, \mathit{premise}\,. \\
\ \textbf{dmatch}\ \mathit{premise} \\
\qquad \neg P \vee \neg Q \Longrightarrow \neg(P \wedge Q)\ \ \textbf{by} \\
\qquad\qquad\qquad \textbf{suppose-absurd}\ H = P \wedge Q\ \textbf{in} \\
\qquad\qquad\qquad\quad \textbf{dlet}\ \mathit{cond}_1 = \neg P \Rightarrow \mathsf{false}\ \textbf{by}\ \ \textbf{assume}\ \neg P\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{absurd}\ \ (\mathbf{!}\mathsf{left\text{-}and}\ \ H)\ \ \neg P \\
\qquad\qquad\qquad\qquad\quad \mathit{cond}_2 = \neg Q \Rightarrow \mathsf{false}\ \textbf{by}\ \ \textbf{assume}\ \neg Q\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{absurd}\ \ (\mathbf{!}\mathsf{right\text{-}and}\ \ H)\ \ \neg Q \\
\qquad\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{!}\mathsf{cd}\ \ \mathit{premise}\ \ \mathit{cond}_1\ \ \mathit{cond}_2
\end{array}
$$

We can now write a more flexible De Morgan's method dm that dispatches the appropriate rule in accordance with the form of the input proposition:

$$
\begin{array}{l}
dm = \mu\, \mathit{premise}\,. \\
\ \textbf{dmatch}\ \mathit{premise} \\
\qquad \neg(P \vee Q) \Longrightarrow \mathbf{!}\,dm_1\ \ \mathit{premise} \\
\qquad \neg P \wedge \neg Q \Longrightarrow \mathbf{!}\,dm_1{}'\ \ \mathit{premise} \\
\qquad \neg(P \wedge Q) \Longrightarrow \mathbf{!}\,dm_2\ \ \mathit{premise} \\
\qquad \neg P \vee \neg Q \Longrightarrow \mathbf{!}\,dm_2{}'\ \ \mathit{premise}
\end{array}
$$

Next we present two methods for the following derived rules of inference:

$$
\frac{(P \wedge Q) \Rightarrow R}{P \Rightarrow (Q \Rightarrow R)}\ [\mathit{curry}] \qquad \frac{P \Rightarrow (Q \Rightarrow R)}{(P \wedge Q) \Rightarrow R}\ [\mathit{uncurry}]
$$

The names *curry* and *uncurry* derive from identifying the connectives $\wedge$ and $\Rightarrow$ with the type constructors $\times$ and $\rightarrow$, whereby the currying rule is viewed as transforming a functional type $(T_1 \times T_2) \rightarrow T_3$ into $T_1 \rightarrow (T_2 \rightarrow T_3)$, with uncurrying proceeding in the converse direction. We have:

```
curry = μ premise .
 dmatch premise
    (P ∧ Q) ⇒ R  ⟹ assume P in
                      assume Q in
                        !mp premise (!both P Q)
```

```
uncurry = μ premise .
 dmatch premise
    P ⇒ (Q ⇒ R)  ⟹ assume P ∧ Q in
                      dlet _ = P by !left-and P ∧ Q
                           _ = Q by !right-and P ∧ Q
                      in
                        !mp (!mp premise P) Q
```

The next method begins to demonstrate the ease with which methods can express powerful derived inference rules. We consider a method that takes a premise of the form $P_1 \Rightarrow (P_2 \Rightarrow (\cdots \Rightarrow (P_n \Rightarrow P_{n+1}) \cdots))$, for *arbitrary* $n > 1$, and derives $(P_1 \wedge \cdots \wedge P_n) \Rightarrow P_{n+1}$:

```
uncurry* = μ premise .
 dmatch premise
    P ⇒ (Q ⇒ R)  ⟹ dlet S = !uncurry premise
                    in
                      dmatch R
                        R1 ⇒ R2 ⟹ !uncurry* S
                        _ ⟹ !claim S
```

We end this section with a method *replace* that takes a proposition P, which must be in the assumption base, as well as two propositions P_{old} and P_{new} such that $P_{old} \Leftrightarrow P_{new}$ is in the assumption base, and derives the equivalence $P \Leftrightarrow P[P_{new}/P_{old}]$, where $P[P_{new}/P_{old}]$ denotes the proposition obtained from P by replacing every occurence of P_{old} by P_{new}. Thus, schematically, we might represent *replace* as follows:

$$\frac{P \qquad P_{old} \Leftrightarrow P_{new}}{P \Leftrightarrow P[P_{new}/P_{old}]}\ [\mathit{replace}]$$

We will need some auxiliary "congruence" methods:

$$\frac{P_1 \Leftrightarrow P_1'}{\neg P_1 \Leftrightarrow \neg P_1'}\ [\textit{not-congruence}] \qquad \frac{P_1 \Leftrightarrow P_1' \quad P_2 \Leftrightarrow P_2'}{P_1 \wedge P_2 \Leftrightarrow P_1' \wedge P_2'}\ [\textit{and-congruence}]$$

and likewise for *or-congruence*, *if-congruence*, and *iff-congruence*. These methods are straightforward; we illustrate with *and-congruence* and leave the rest as an exercise:

$$
\begin{array}{l}
\textit{and-congruence} = \mu\, \textit{equiv}_1, \textit{equiv}_2\, . \\
\ \ \mathbf{dmatch}\ [\textit{equiv}_1, \textit{equiv}_2] \\
\qquad [P_1 \Leftrightarrow P_1', P_2 \Leftrightarrow P_2'] \Longrightarrow \mathbf{dlet}\ \textit{cond}_1 = \mathbf{assume}\ P_1 \wedge P_2\ \mathbf{in} \\
\qquad\qquad \mathbf{dlet}\ \_ = P_1'\ \mathbf{by}\ !\mathsf{mp}\ (!\mathsf{left\text{-}iff}\ \ \textit{equiv}_1) \\
\qquad\qquad\qquad\qquad (!\mathsf{left\text{-}and}\ \ P_1 \wedge P_2) \\
\qquad\qquad\qquad \_ = P_2'\ \mathbf{by}\ !\mathsf{mp}\ (!\mathsf{left\text{-}iff}\ \ \textit{equiv}_2) \\
\qquad\qquad\qquad\qquad (!\mathsf{right\text{-}and}\ \ P_1 \wedge P_2) \\
\qquad\qquad \mathbf{in} \\
\qquad\qquad\quad !\mathsf{both}\ \ P_1'\ \ P_2' \\
\qquad\quad \textit{cond}_2 = \mathbf{assume}\ P_1' \wedge P_2'\ \mathbf{in} \\
\qquad\qquad \mathbf{dlet}\ \_ = P_1\ \mathbf{by}\ !\mathsf{mp}\ (!\mathsf{right\text{-}iff}\ \ \textit{equiv}_1) \\
\qquad\qquad\qquad\qquad (!\mathsf{left\text{-}and}\ \ P_1' \wedge P_2') \\
\qquad\qquad\qquad \_ = P_2\ \mathbf{by}\ !\mathsf{mp}\ (!\mathsf{right\text{-}iff}\ \ \textit{equiv}_2) \\
\qquad\qquad\qquad\qquad (!\mathsf{right\text{-}and}\ \ P_1' \wedge P_2') \\
\qquad\qquad \mathbf{in} \\
\qquad\qquad\quad !\mathsf{both}\ \ P_1\ \ P_2 \\
\qquad \mathbf{in} \\
\qquad\quad P_1 \wedge P_2 \Leftrightarrow P_1' \wedge P_2'\ \mathbf{by}\ !\mathsf{equiv}\ \ \textit{cond}_1\ \ \textit{cond}_2
\end{array}
$$

where we assume the straightforward extension of having lists of constants $[c_1, \ldots, c_n]$ as primitive values, and the ability to match such lists against appropriate patterns.

We can now express *replace* as follows:

$$
\begin{array}{l}
\textit{replace} = \mu\, P, P_{\textit{old}}, P_{\textit{new}}\, . \\
\ \ \mathbf{dcond}\ (=\ \ P\ \ P_{\textit{old}}) \\
\qquad !\mathsf{claim}\ \ P_{\textit{old}} \Leftrightarrow P_{\textit{new}} \\
\qquad \mathbf{dmatch}\ P \\
\qquad\quad \neg Q \Longrightarrow \mathbf{dlet}\ \textit{equiv} = !\textit{replace}\ \ Q\ \ P_{\textit{old}}\ \ P_{\textit{new}} \\
\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad !\textit{not-congruence}\ \ \textit{equiv} \\
\qquad\quad P_1\ \textit{op}\ P_2 \Longrightarrow \mathbf{dlet}\ \textit{equiv}_1 = !\textit{replace}\ \ P_1\ \ P_{\textit{old}}\ \ P_{\textit{new}} \\
\qquad\qquad\qquad\qquad \textit{equiv}_2 = !\textit{replace}\ \ P_2\ \ P_{\textit{old}}\ \ P_{\textit{new}} \\
\qquad\qquad\quad \mathbf{in}
\end{array}
$$

$$\begin{array}{l} \quad\quad\quad !\,\mathit{congruence}\ \ \mathit{equiv}_1\ \ \mathit{equiv}_2\ \ \mathit{op} \\ \_\ \Longrightarrow\ !\,\mathit{reflex}\ \ P \end{array}$$

where *reflex* is a trivial unary method that derives the tautology $P \Leftrightarrow P$ for any given P, and where

$\mathit{congruence} = \mu\, \mathit{equiv}_1, \mathit{equiv}_2, \mathit{op}\,.$
dmatch *op*

$$\begin{array}{lll} \mathsf{and} & \Longrightarrow & !\,\mathit{and\text{-}congruence}\ \ \mathit{equiv}_1\ \ \mathit{equiv}_2 \\ \mathsf{or} & \Longrightarrow & !\,\mathit{or\text{-}congruence}\ \ \mathit{equiv}_1\ \ \mathit{equiv}_2 \\ \mathsf{if} & \Longrightarrow & !\,\mathit{if\text{-}congruence}\ \ \mathit{equiv}_1\ \ \mathit{equiv}_2 \\ \mathsf{iff} & \Longrightarrow & !\,\mathit{iff\text{-}congruence}\ \ \mathit{equiv}_1\ \ \mathit{equiv}_2 \end{array}$$

Note that because the propositional constructors and, or, etc., are data values (constants), we can match propositions against patterns such as $P_1\ \mathit{op}\ P_2$, where *op* is a pattern variable that can become bound to a propositional constructor. For instance, matching $A_2 \vee (A_4 \Rightarrow A_8)$ against the pattern $P_1\ \mathit{op}\ P_2$ would bind P_1 to the atom A_2, *op* to the constructor or, and P_2 to the proposition $A_4 \Rightarrow A_8$. This type of pattern-matching on propositions has been implemented in Athena [3], and has been found to result in very succinct method and function definitions.

9.5 A sequent calculus for natural deduction as a pure $\lambda\mu$ system

Definition

In this section we will present a sequent-based natural deduction system for first-order logic as a pure $\lambda\mu$ system. Here the set of constants C includes every constant, function, and relation symbol of some first-order logic signature $\mathbf{\Omega}$, every variable[3] x in some countably infinite set of variables $\mathcal{V}$, and every term t and formula F that can be built from these symbols and variables (see Section 6.1). Formulas, in particular, are generated by the following grammar:

$$F ::= R(t_1, \ldots, t_n) \mid \mathsf{true} \mid \mathsf{false} \mid \neg F \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid F \Leftrightarrow G \mid (\forall x)F \mid (\exists x)F$$

where R ranges over relation symbols of arity n. As usual, we identify alphabetically equivalent formulas; we write $\{x \mapsto t\}\, F$ for the formula obtained from F by replacing every free occurence of x by t (this is always safe with an appropriate α-conversion);

[3] It is an important distinction that in this approach object-level variables are treated as *constants* at the meta level.

and we write $FV(F)$ ($FV(\Gamma)$, for a set of formulas Γ) for the set of variables that occur free in F (in the members of Γ). As constants we also take finite sets of formulas, as well as all pairs of the form $\langle\Gamma, F\rangle$, where Γ is a finite set of formulas and F is a single formula. Such pairs are called *sequents*, and will serve as the sentences of the system.

In addition, we will need the following constants, which will serve as inference rules: init, dilute, true-axiom, false-axiom, mp, discharge, neg-intro, neg-elim, conj-intro, conj-elim-l, conj-elim-r, disj-elim, disj-intro-l, disj-intro-r, bicond-intro, bicond-elim-l, cut, bicond-elim-r, uspec, ugen, espec, and egen. The cut rule is not necessary for forward proof (it does not affect completeness), but we will see that it is useful for backwards proof search. The semantics of these constants are given by the δ-rules shown in Figure 9.3.

Example proofs and methods (forward style)

The sentences of this system are sequents, thus a sequent $\langle\Gamma, F\rangle$ is *derivable* in (or from) an assumption base β iff there is a deduction D such that $\beta \vdash D \rightsquigarrow \langle\Gamma, F\rangle$; and $\langle\Gamma, F\rangle$ is provable iff it is derivable from the empty assumption base, i.e., iff there is a deduction D such that $\emptyset \vdash D \rightsquigarrow \langle\Gamma, F\rangle$. A sequent $\langle\Gamma, F\rangle$ is *sound* iff $\Gamma \models F$, i.e., iff F is a logical consequence of Γ. It is not difficult to show that if an assumption base β contains only sound sequents, then every sequent derivable from β is sound as well. In particular, then, all provable sequents are sound. Finally, in this framework we say that a formula F is *deducible* from a finite set of formulas Γ iff the sequent $\langle\Gamma, F\rangle$ is provable. Hence, from what we said above, deducibility is sound: if F is deducible from Γ then it follows logically from it. Completeness means that if F follows logically from Γ then it is also deducible from it. This system is complete as well, although we will not prove that.

We continue with some example deductions, beginning with our running example $F \wedge G \Rightarrow G \wedge F$. Specifically, we prove that the sequent $\langle\emptyset, F \wedge G \Rightarrow G \wedge F\rangle$ is provable by presenting a deduction D such that $\emptyset \vdash D \rightsquigarrow \langle\emptyset, F \wedge G \Rightarrow G \wedge F\rangle$. We present D in conclusion-annotated form:

dlet $S_1 = \langle\{F \wedge G\}, F \wedge G\rangle$ **by** !init $F \wedge G$
$S_2 = \langle\{F \wedge G\}, G\rangle$ **by** !conj-elim-r S_1
$S_3 = \langle\{F \wedge G\}, F\rangle$ **by** !conj-elim-l S_1
$S_4 = \langle\{F \wedge G\}, G \wedge F\rangle$ **by** !conj-intro S_2 S_3
in
$\langle\emptyset, F \wedge G \Rightarrow G \wedge F\rangle$ **by** !discharge S_4 $F \wedge G$

Here it is without annotated conclusions:

$$\emptyset \vdash \mathbf{dapp}(\mathsf{init}, F) \rightsquigarrow \langle \{F\}, F\rangle$$
$$\{\langle \Gamma, F\rangle\} \vdash \mathbf{dapp}(\mathsf{dilute}, \langle \Gamma, F\rangle, G) \rightsquigarrow \langle \Gamma \cup \{G\}, F\rangle$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{true\text{-}axiom},) \rightsquigarrow \langle \emptyset, \mathsf{true}\rangle$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{false\text{-}axiom},) \rightsquigarrow \langle \emptyset, \neg\mathsf{false}\rangle$$
$$\{\langle \Gamma \cup \{F\}, G\rangle, \langle \Gamma \cup \{F\}, \neg G\rangle\} \vdash \mathbf{dapp}(\mathsf{neg\text{-}intro}, \langle \Gamma \cup \{F\}, G\rangle, \langle \Gamma \cup \{F\}, \neg G\rangle, F) \rightsquigarrow \langle \Gamma, \neg F\rangle$$
$$\{\langle \Gamma, \neg\neg F\rangle\} \vdash \mathbf{dapp}(\mathsf{neg\text{-}elim}, \langle \Gamma, \neg\neg F\rangle) \rightsquigarrow \langle \Gamma, F\rangle$$
$$\{\langle \Gamma \cup \{F\}, G\rangle\} \vdash \mathbf{dapp}(\mathsf{discharge}, \langle \Gamma \cup \{F\}, G\rangle, F) \rightsquigarrow \langle \Gamma, F \Rightarrow G\rangle$$
$$\{\langle \Gamma, F \Rightarrow G\rangle, \langle \Gamma, F\rangle\} \vdash \mathbf{dapp}(\mathsf{mp}, \langle \Gamma, F \Rightarrow G\rangle, \langle \Gamma, F\rangle) \rightsquigarrow \langle \Gamma, G\rangle$$
$$\{\langle \Gamma, F\rangle, \langle \Gamma, G\rangle\} \vdash \mathbf{dapp}(\mathsf{conj\text{-}intro}, \langle \Gamma, F\rangle, \langle \Gamma, G\rangle) \rightsquigarrow \langle \Gamma, F \wedge G\rangle$$
$$\{\langle \Gamma, F \wedge G\rangle\} \vdash \mathbf{dapp}(\mathsf{conj\text{-}elim\text{-}l}, \langle \Gamma, F \wedge G\rangle) \rightsquigarrow \langle \Gamma, F\rangle$$
$$\{\langle \Gamma, F \wedge G\rangle\} \vdash \mathbf{dapp}(\mathsf{conj\text{-}elim\text{-}r}, \langle \Gamma, F \wedge G\rangle) \rightsquigarrow \langle \Gamma, G\rangle$$
$$\{\langle \Gamma \cup \{F_1\}, G\rangle, \langle \Gamma \cup \{F_2\}, G\rangle\} \vdash \mathbf{dapp}(\mathsf{disj\text{-}elim}, \langle \Gamma \cup \{F_1\}, G\rangle, \langle \Gamma \cup \{F_2\}, G\rangle) \rightsquigarrow \langle \Gamma \cup \{F_1 \vee F_2\}, G\rangle$$
$$\{\langle \Gamma, F\rangle\} \vdash \mathbf{dapp}(\mathsf{disj\text{-}intro\text{-}l}, \langle \Gamma, F\rangle, G) \rightsquigarrow \langle \Gamma, F \vee G\rangle$$
$$\{\langle \Gamma, F\rangle\} \vdash \mathbf{dapp}(\mathsf{disj\text{-}intro\text{-}r}, \langle \Gamma, F\rangle, G) \rightsquigarrow \langle \Gamma, G \vee F\rangle$$
$$\{\langle \Gamma, F \Rightarrow G\rangle, \langle \Gamma, G \Rightarrow F\rangle\} \vdash \mathbf{dapp}(\mathsf{bicond\text{-}intro}, \langle \Gamma, F \Rightarrow G\rangle, \langle \Gamma, G \Rightarrow F\rangle) \rightsquigarrow \langle \Gamma, F \Leftrightarrow G\rangle$$
$$\{\langle \Gamma, F \Leftrightarrow G\rangle\} \vdash \mathbf{dapp}(\mathsf{bicond\text{-}elim\text{-}l}, F \Leftrightarrow G) \rightsquigarrow \langle \Gamma, F \Rightarrow G\rangle$$
$$\{\langle \Gamma, F \Leftrightarrow G\rangle\} \vdash \mathbf{dapp}(\mathsf{bicond\text{-}elim\text{-}r}, F \Leftrightarrow G) \rightsquigarrow \langle \Gamma, G \Rightarrow F\rangle$$
$$\{\langle \Gamma, F_1\rangle, \ldots, \langle \Gamma, F_n\rangle, \langle \Gamma \cup \{F_1, \ldots, F_n\}, G\rangle\} \vdash \mathbf{dapp}(\mathsf{cut}, \Gamma, \{F_1, \ldots, F_n\}, G) \rightsquigarrow \langle \Gamma, G\rangle$$
$$\{\langle \Gamma, (\forall\, x)\, F\rangle\} \vdash \mathbf{dapp}(\mathsf{uspec}, \langle \Gamma, (\forall\, x)\, F\rangle, t) \rightsquigarrow \langle \Gamma, \{x \mapsto t\}\, F\rangle$$
$$\{\langle \Gamma, F\rangle\} \vdash \mathbf{dapp}(\mathsf{ugen}, \langle \Gamma, F\rangle, x) \rightsquigarrow \langle \Gamma, (\forall\, x)\, F\rangle, \text{ provided } x \notin FV(\Gamma).$$
$$\{\langle \Gamma, \{x \mapsto t\}\, F\rangle\} \vdash \mathbf{dapp}(\mathsf{egen}, \langle \Gamma, (\exists\, x)\, F\rangle) \rightsquigarrow \langle \Gamma, (\exists\, x)\, F\rangle$$
$$\{\langle \Gamma, (\exists\, x)\, F\rangle, \langle \Gamma \cup \{\{x \mapsto y\}\, F\}, G\rangle\} \vdash \mathbf{dapp}(\mathsf{espec}, \langle \Gamma, (\exists\, x)\, F\rangle, y, G) \rightsquigarrow \langle \Gamma, G\rangle$$
provided y does not occur in Γ, F, or G.

Figure 9.3: δ-axioms for the constants of the sequent calculus.

dlet $S_1 = \mathbf{!}\mathsf{init}\ \ F \wedge G$
$\quad S_2 = \mathbf{!}\mathsf{conj\text{-}elim\text{-}r}\ \ S_1$
$\quad S_3 = \mathbf{!}\mathsf{conj\text{-}elim\text{-}l}\ \ S_1$
$\quad S_4 = \mathbf{!}\mathsf{conj\text{-}intro}\ \ S_2\ \ S_3$
in
$\quad \mathbf{!}\mathsf{discharge}\ \ S_4\ \ F \wedge G$

or in even shorter form:

```
(!dishcarge  (!conj-intro  (!conj-elim-r (!init F ∧ G))
                           (!conj-elim-r (!init F ∧ G)))
             F ∧ G)
```

Compare all three of the above with the following $\lambda\mu$–$\mathcal{CND}_0$ deduction:

assume $P \wedge Q$ **in**
  !both (!right-and $P \wedge Q$)
     (!left-and $P \wedge Q$)

or even with its $\mathcal{CND}$ equivalent:

assume $F \wedge G$ **in**
  begin
    right-and $F \wedge G$;
    left-and $F \wedge G$;
    both G, F
  end

It is evident that the last two deductions are more clear and to the point than their sequent-based counterparts.

Next, here is a deduction D that proves the sequent

$$\langle \emptyset, (\forall\, x)\, [P(x) \wedge Q(x)] \Rightarrow (\forall\, x)\, P(x) \wedge (\forall\, x)\, Q(x) \rangle :$$

dlet $S_1 =$!init $(\forall\, x)\, [P(x) \wedge Q(x)]$
  $S_2 =$!uspec $s1$ y
  $S_3 =$!conj-elim-l $s2$
  $S_4 =$!ugen S_3 y
  $S_5 =$!conj-elim-r $s2$
  $S_6 =$!ugen S_5 y
  $S_7 =$!conj-intro S_4 S_6
in
  !discharge S_7 $(\forall\, x)\, [P(x) \wedge Q(x)]$

And here it is in conclusion-annotated form:

$$
\begin{array}{ll}
\textbf{dlet} & S_1 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, (\forall\, x)\,[P(x) \wedge Q(x)]\rangle \textbf{ by } !\mathsf{init}\ \ (\forall\, x)\,[P(x) \wedge Q(x)] \\
& S_2 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, P(y) \wedge Q(y)\rangle \textbf{ by } !\mathsf{uspec}\ \ s1\ \ y \\
& S_3 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, P(y)\rangle \textbf{ by } !\mathsf{conj\text{-}elim\text{-}l}\ \ s2 \\
& S_4 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, (\forall\, y)\,P(y)\rangle \textbf{ by } !\mathsf{ugen}\ \ S_3\ \ y \\
& S_5 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, Q(y)\rangle \textbf{ by } !\mathsf{conj\text{-}elim\text{-}r}\ \ s2 \\
& S_6 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, (\forall\, y)\,Q(y)\rangle \textbf{ by } !\mathsf{ugen}\ \ S_5\ \ y \\
& S_7 = \langle\{(\forall\, x)\,[P(x) \wedge Q(x)]\}, (\forall\, y)\,P(y) \wedge (\forall\, y)\,Q(y)\rangle \textbf{ by } !\mathsf{conj\text{-}intro}\ \ S_4\ \ S_6 \\
\textbf{in} & \\
\multicolumn{2}{l}{\quad \langle\emptyset, (\forall\, x)\,[P(x) \wedge Q(x)] \Rightarrow (\forall\, x)\,P(x) \wedge (\forall\, x)\,Q(x)\rangle \textbf{ by } !\mathsf{discharge}\ \ S_7\ \ (\forall\, x)\,[P(x) \wedge Q(x)]}
\end{array}
$$

Insofar perspicuity is concerned, the second deduction is more verbose, but easier to follow than the first. Contrast both deductions with the proof of the same result in $\mathcal{CND}$ (Section 6.3), or with the $\lambda\mu$–$\mathcal{CND}_1$ proof of it (Section 9.7). We believe the comparisons will convince the reader that sequents are not a good medium for *presenting* proofs, as we argued earlier (Section 7.2.2). Proposition-based natural deduction mechanisms such as **assume**, **pick-any**, etc., are far more effective in bringing out the structure of the proof in a lucid form.

Our last two examples use pattern matching (**dmatch** deductions). That subject is discussed in Section 8.6.6, and although the details of that discussion are specific to a particular system, the basic ideas are applicable in a variety of contexts. For our present purposes we do not need to specify a pattern language in detail or show how to desugar **dmatch** deductions. Rather, we will rely on the reader's intuitive understanding of pattern matching, bolstered by the discussion in Section 8.6.6. One assumption we will make explicit is that multiple occurences of the same identifier in a (non-linear) pattern must be bound to the same value during matching.

Let us now write a recursive method that weakens (dilutes) the context of a given sequent with an arbitrary number of formulas. The built-in binary method dilute can only insert a formula at a time: it takes a sequent $\langle\Gamma, F\rangle$ and a single formula G and returns the sequent $\langle\Gamma \cup \{G\}, F\rangle$. It would be convenient to have a binary method *dilute** that takes a sequent $\langle\Gamma, F\rangle$ and a *set* of formulas Δ and derives the sequent $\langle\Gamma \cup \Delta, F\rangle$. It is not necessary to take such a method as primitive; we can express it in terms of the simpler primitive dilute as follows:

$$
\begin{array}{l}
\mathit{dilute}^* = \mu\, S, \Delta\,. \\
\quad \textbf{dmatch } \Delta \\
\qquad \emptyset \implies !\mathsf{claim}\ \ S \\
\qquad \Delta' \cup \{F\} \implies !\,\mathit{dilute}^*\ (!\mathsf{dilute}\ \ S\ \ F)\ \ \Delta'
\end{array}
$$

As an example, the reader will verify that the method call

$$!\,dilute^*\ (!\mathsf{init}\ \ F)\ \ \{G_1, G_2, G_3\}$$

produces the sequent $\langle \{F, G_1, G_2, G_3\}, F\rangle$, in any assumption base.

We can now write a method that derives any sequent of the form $\langle \Gamma \cup \{F\}, F\rangle$ as follows:

$reflex = \mu\, F, \Gamma\,.$
  $!\,dilute^*\ (!\mathsf{init}\ \ F)\ \ \Gamma$

Thus *reflex* takes a formula F and a set of formulas Γ and derives $\langle \Gamma \cup \{F\}, F\rangle$.

For our final example we will assume that the system is equipped with lists. We note that any $\lambda\mu$ system $(kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n); C; \Delta)$ without lists can be conservatively extended to a system $(kwd_1(\overrightarrow{\Xi}_1), \ldots, kwd_n(\overrightarrow{\Xi}_n); C'; \Delta')$ with lists as follows. Let cons, head, tail, nil, and list be five constants that do not occur in C, and define C' as follows: every constant $c \in C$ is in C'; and if $c_1, \ldots, c_n$ are in C' then so is the term $\mathsf{list}(c_1, \ldots, c_n)$ (denoting the list $[c_1, \ldots, c_n]$). We also stipulate that C' contains the five constructors mentioned above. Finally, the δ-rules of the new system include all of the old rules plus the ones that give the desired semantics to the new constructors, e.g.,

$$\begin{gathered}\emptyset \vdash \mathbf{app}(\mathsf{nil}) \rightsquigarrow \mathsf{list}()\\ \emptyset \vdash \mathbf{app}(\mathsf{cons}, c, \mathsf{list}(c_1, \ldots, c_n)) \rightsquigarrow \mathsf{list}(c, c_1, \ldots, c_n)\\ \emptyset \vdash \mathbf{app}(\mathsf{head}, \mathsf{list}(c_1, \ldots, c_n)) \rightsquigarrow c_1\end{gathered}$$

and so on. In what follows we will write $[c_1, \ldots, c_n]$ as an abbreviation for $\mathsf{list}(c_1, \ldots, c_n)$.

With this background, let us write a method *Tran* that takes two sequents $S_1 = \langle \Gamma, F \Rightarrow G\rangle$ and $S_2 = \langle \Gamma, G \Rightarrow H\rangle$ that are assumed to hold (i.e., we assume that S_1 and S_2 are in the assumption base at the time when *Tran* is invoked), and produces the sequent $\langle \Gamma, F \Rightarrow H\rangle$:

$Tran = \mu\, S_1, S_2\,.$
  dmatch $[S_1, S_2]$
    $[\langle \Gamma, F \Rightarrow G\rangle, \langle \Gamma, G \Rightarrow H\rangle] \Longrightarrow$
      dlet $S_3 = !\mathsf{dilute}\ \ S_1\ \ F$
        $S_4 = !\mathsf{dilute}{*}\ \ (!\mathsf{init}\ \ F)\ \ \Gamma$
        $S_5 = !\mathsf{dilute}\ \ S_2\ \ F$
        $S_6 = !\mathsf{mp}\ \ S_3\ \ S_4$
        $S_7 = !\mathsf{mp}\ \ S_5\ \ S_6$
      in
        $!\mathsf{discharge}\ \ S_7\ \ F$

For instance, the reader will verify that

$$\{\langle\emptyset, (\forall\, x)\, P(x) \Rightarrow (\forall\, x)\, Q(x)\rangle, \langle\emptyset, (\forall\, x)\, Q(x) \Rightarrow (\exists\, x)\, R(x)\rangle\} \vdash$$
$$\mathbf{!}\, \mathit{Tran}\ \langle\emptyset, (\forall\, x)\, P(x) \Rightarrow (\forall\, x)\, Q(x)\rangle\ \ \langle\emptyset, (\forall\, x)\, Q(x) \Rightarrow (\exists\, x)\, R(x)\rangle \rightsquigarrow \langle\emptyset, (\forall\, x)\, P(x) \Rightarrow (\exists\, x)\, R(x)\rangle.$$

Examples of proof search (backwards style)

In this section we will write a theorem prover for the $\{\Rightarrow, \wedge\}$–fragment of this logic. Accordingly, we will only "recognize" conditionals and conjunctions; everything else will count as an atom. The prover will be written as a method. The advantage of this, as we have pointed out, is that methods are guaranteed to give sound results once the primitive constants have been shown to be sound; there is no need to introduce a type system for the purpose of ensuring soundness. A method call is a deduction, and every deduction (immediately recognizable as such by its syntactic form) produces sound results.

Let a goal sequent $\langle\Gamma, F\rangle$ be given, and let us assume that we have a method *prove-atom* which we can apply if F is an atom. This will be our "base case". If F is not an atom then we decompose it and generate appropriate subgoals which we will try to satisfy recursively. Once the subgoals have been established, we need to combine them in the right way in order to derive the desired goal. In the HOL world this means applying the "justification function" produced by the generation of the subgoals to the subgoal theorems. In our case there is no need to generate justification functions and to keep track of the subgoals in a stack; all of this will automatically be managed by the semantics of the language—and in particular by the semantics of nested method calls. The call tree spawned by the top recursive method call *is* the subgoal stack. More specifically, the method proceeds in accordance with the structure of F:

- If F is a conjunction $F_1 \wedge F_2$, recursively prove the subgoals $\langle\Gamma, F_1\rangle$ and $\langle\Gamma, F_2\rangle$, and then use conj-intro to derive the goal.

- If F is a conditional $F_1 \Rightarrow F_2$, recursively prove the subgoal $\langle\Gamma \cup \{F_1\}, F_2\rangle$ and then discharge F_1 to generate the desired goal.

- Finally, if F is an atom (i.e., neither a conjunction nor a conditional), then apply *prove-atom* to it.

Thus we arrive at the following method, which we call *prove*:

prove $= \mu\, \mathit{goal}.$
  dmatch *goal*
    $\langle\Gamma, F \wedge G\rangle \implies$ **dlet** *left* $= \mathbf{!}\,\mathit{prove}\ \langle\Gamma, F\rangle$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad right = \mathbf{!}\,prove\ \langle \Gamma, G\rangle \\
\qquad\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{!}\textsf{conj-intro}\ \ left\ \ right \\
\langle \Gamma, F \Rightarrow G\rangle \implies \mathbf{dlet}\ \ S\ = \mathbf{!}\,prove\ \langle \Gamma \cup \{F\}, G\rangle \\
\qquad\qquad\qquad\qquad\quad \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{!}\textsf{discharge}\ \ S\ \ F \\
\_ \implies \mathbf{!}\,prove\text{-}atom\ \ goal
\end{array}
$$

We must now implement *prove-atom*, whose task is to prove sequents of the form $\langle \Gamma, A\rangle$, where A is an atom. The basic idea is to keep applying Modus Ponens and conjunction eliminations in the context Γ until we arrive at a sequent of the form $\langle \Gamma' \cup \{A\}, A\rangle$. For instance, suppose that the goal is $\langle \{A \Rightarrow B, B \Rightarrow C, A\}, C\rangle$, for atoms A, B, and C. If we apply Modus Ponens to the context once, we obtain B; if we apply Modus Ponens again to the result, we get the sequent

$$\langle \{A \Rightarrow B, B \Rightarrow C, A, B, C\}, C\rangle$$

which clearly holds. The only question is how to justify going (backwards) from the last sequent to the original goal $\langle \{A \Rightarrow B, B \Rightarrow C, A\}, C\rangle$. We can do this by recursively applying the cut rule. In general, if we have obtained

$$\langle \{F \Rightarrow G, F, G\} \cup \Gamma, H\rangle$$

then we can derive

$$\langle \{F \Rightarrow G, F\} \cup \Gamma, H\rangle$$

through the cut rule by proving

$$\langle \{F \Rightarrow G, F\} \cup \Gamma, G\rangle$$

thereby showing that G is not really necessary. Likewise, if we have shown

$$\langle \{F \wedge G, F, G\} \cup \Gamma, H\rangle$$

then we can use the cut method to derive

$$\langle \{F \wedge G\} \cup \Gamma, H\rangle$$

by proving the sequents

$$\langle \{F \wedge G\} \cup \Gamma, F\rangle \ \ \text{and} \ \ \langle \{F \wedge G\} \cup \Gamma, G\rangle$$

which is straightforward. Thus we arrive at the following:

$$
\begin{array}{l}
\textit{prove-atom} = \mu\,\langle \Gamma, A\rangle\,.\\
\quad \textbf{dmatch}\;\; A \in \Gamma\\
\quad\quad \mathsf{true} \Longrightarrow (\,!\,\textit{reflex}\;\; A\;\; \Gamma)\\
\quad\quad \mathsf{false} \Longrightarrow \textbf{dmatch}\;\; (\textit{detach-if?}\;\; \Gamma)\\
\qquad\qquad\qquad [F \Rightarrow G, F] \Longrightarrow \textbf{dlet}\;\; \_ = \,!\,\textit{prove-atom}\;\; \langle \Gamma \cup \{G\}, A\rangle\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; \_ = \,!\,\mathsf{mp}\;\; (\,!\,\textit{reflex}\;\; F \Rightarrow G\;\; \Gamma)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; (\,!\,\textit{reflex}\;\; F\;\; \Gamma)\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; !\,\mathsf{cut}\;\; \Gamma\;\; \{G\}\;\; A\\
\qquad\qquad\qquad [] \Longrightarrow \textbf{dmatch}\;\; (\textit{detach-and?}\;\; \Gamma)\\
\qquad\qquad\qquad\qquad\quad [F \wedge G] \Longrightarrow \textbf{dlet}\;\; \_ = \,!\,\textit{prove-atom}\;\; \langle \Gamma \cup \{F, G\}, A\rangle\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ = \,!\,\mathsf{conj\text{-}elim\text{-}l}\;\; (\,!\,\textit{reflex}\;\; F \wedge G\;\; \Gamma)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \_ = \,!\,\mathsf{conj\text{-}elim\text{-}r}\;\; (\,!\,\textit{reflex}\;\; F \wedge G\;\; \Gamma)\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad !\,\mathsf{cut}\;\; \Gamma\;\; \{F, G\}\;\; A
\end{array}
$$

Note that we have written the argument of *prove-atom* as a pattern (in ML style). This is just a convenient shorthand; for any pattern π, a method definition of the form $I = \mu\,\pi\,.\,D$ should be understood as syntax sugar for $I = \mu\,S\,.\,\textbf{dmatch}\;\; S\;\; \pi \Longrightarrow\;\; D$.

The method *prove-atom* uses two unary functions *detach-if?* and *detach-and?*. The former takes a set of formulas Γ as input and checks to see if Γ contains two formulas of the form $F \Rightarrow G$ and G, where G *is not in* Γ. If so, *detach-if?* returns the two-element list $[F \Rightarrow G, F]$; otherwise it returns the empty list $[]$. Note that the condition that G is not a member of Γ is crucial in preventing the method from getting into an infinite loop. If we do not observe this caveat then it is possible for the method call $(\,!\,\textit{prove-atom}\;\; \langle\{A \Rightarrow B, B \Rightarrow C, A\}, C\rangle)$ to diverge, indefinitely applying Modus Ponens to $A \Rightarrow B$ and A and deriving B. Likewise, *detach-and?* takes a set of formulas Γ and checks to see if it contains a formula of the form $F \wedge G$ such that either $F \notin \Gamma$ or $G \notin \Gamma$. If so, *detach-and?* returns the single-element list $[F \wedge G]$; otherwise it returns the empty list $[]$. Again, it is important to ensure that not both of F and G are already in the list, in order to avoid duplicate work (and possibly getting into an infinite loop). These two functions are clearly easy to implement and we omit their definitions.

Figure 9.4 depicts the flow of control during the evaluation of

$$!\,\textit{prove}\;\; \langle \emptyset, F \wedge G \Rightarrow G \wedge F\rangle.$$

Downward arrows correspond to recursive method calls, while upward arrows correspond to primitive method applications. Thus we see that recursion is used analytically

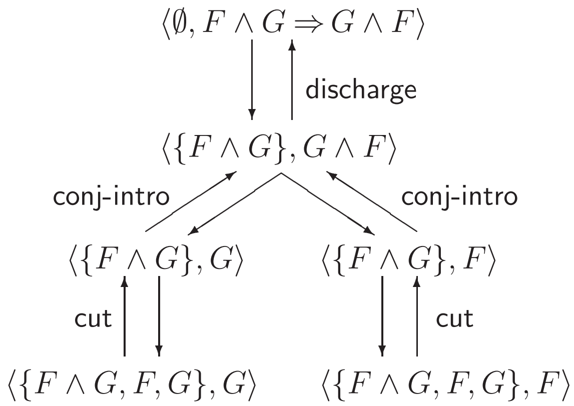


Figure 9.4: The call tree for the method application $\mathbf{!}\,prove\ \langle \emptyset, F \wedge G \Rightarrow G \wedge F \rangle$.

("backwards") for proof search, attempting to decompose the goal into axioms, while primitive method applications work synthetically in the reverse direction ("forward"), building up the goal from the axioms, thus ultimately justifying the preceding recursive decomposition. The reader should compare this example with that shown in Figure 9.8.

9.6 Unification as deduction

In this section we set up a $\lambda\mu$ system in which one can prove sentences of the form "E is unifiable", where E is a finite system of equations between Herbrand terms. Based on this logic, which is shown to be sound, we formulate a unification procedure as a *method*. We also prove that the logic is complete.

9.6.1 Unification via transformations

For the remainder of this section, fix a set $\mathcal{F}$ of function symbols and a disjoint set $\mathcal{V}$ of variables (see Appendix A for the relevant terminology and notation). In what follows, by a "term" we will mean a Herbrand term over $\mathcal{F}$ and $\mathcal{V}$; and by an "equation" we will mean an ordered pair of terms $\langle s, t \rangle$, which will be more suggestively written as $s \approx t$.

Recall that two terms s and t are unifiable iff there exists a substitution θ such that $\overline{\theta}(s) = \overline{\theta}(t)$; we say that such a θ "unifies" s and t. Unification lies at the heart of many symbolic-reasoning engines, e.g., the Prolog abstract machine, resolution theorem provers, Hindley-Milner type-inference systems such as found in ML, and others. The following problem is mechanically solvable: given two terms s and t, determine whether they are unifiable; if they are, produce a most general unifying substitution for them. Robinson [63] was the first researcher who explicitly formulated this problem, gave an algorithm for it, and proved the algorithm's correctness. Although his algorithm performs fairly in practice, it has exponential worst-case complexity, both in time and in space. Subsequent work resulted in major efficiency improvements, and there are now several unification algorithms of low polynomial time and space complexity [56, 48, 20] (see Knight's survey [44] for an overview). Most of these algorithms, however, need to maintain elaborate data structures and their details are fairly intricate, in contrast to Robinson's original procedure.

Martelli and Montinari in 1982 [49] introduced a particularly elegant and clean formulation of unification based on a set of transformations akin to those used by Gauss-Jordan elimination in solving systems of linear equations. Although they were the first ones to discuss such transformations explicitly in the context of unification, the basic ideas were already present in Herbrand's thesis in the 1930s. We quote from "Logical writings of Herbrand" [37], page 148:

> Now, to find an appropriate set of associated equations is easy, if such a set exists; it suffices, for each system of equations between arguments, to proceed by recursion, using one of the following procedures which simplify the system of equations to be satisfied.
>
> (1) If one of the equations to be satisfied equates a restricted variable x to an individual, either this individual contains x, and then the equations cannot be satisfied, or else the individual does not contain x, and then the equation will be one of the associated equations that we are looking for; in the other equations to be satisfied we replace x by the individual;
>
> (2) If one of the equations to be satisfied equates a general variable to an individual that is not a restricted variable, the equation cannot be satisfied;
>
> (3) If one of the equations to be satisfied equates
>
> $$f_1(\phi_1, \ldots, \phi_n) \quad \text{to} \quad f_2(\psi_1, \ldots, \psi_n),$$
>
> either the elementary functions f_1 and f_2 are different, and then the equation cannot be satisfied, or they are the same, and then we turn to those equations that equate the ϕ_i to the ψ_i.
>
> Therefore, if we successively consider each prenex form of P, we shall be able, after a finite and determinate number of steps, to decide whether the proposition P is a normal identity.

As we will see, this is the crux of the Martelli-Montanari algorithm. In what follows we will use the acronym HMM as an abbreviation for "Herbrand-Martelli-Montanari".

The HMM algorithm approaches unification from a more general angle than other procedures, dealing with finite *systems* of equations rather than with single equations. By a system of equations we will mean a multiset of the form

$$E = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}. \tag{9.37}$$

Because this is a multiset, an equation might have multiple occurences in E, and those occurences are significant.[4] We will write $E, s \approx t$ as an abbreviation for the multiset union of E and $\{s \approx t\}$.[5]

A system of equations E of the form 9.37 is *unifiable* iff there exists a substitution θ that unifies every equation in E, i.e., such that $\overline{\theta}(s_i) = \overline{\theta}(t_i)$ for $i = 1, \ldots, n$. We call θ a *unifier* of E. If $\sigma \leq \theta$ for every σ that unifies E then we say that θ is a *most general unifier* ("mgu") of E. Most general unifiers are unique up to $\equiv$ (composition with a renaming, see Section A.2.2), and in that sense we may speak of *the* mgu of some E. We write $U(E)$ for the set of all unifiers of E, where we might of course have $U(E) = \emptyset$ if E is not unifiable. Thus the traditional unification problem of determining whether two given terms s and t can be unified is reducible to the problem of deciding whether the system $\{s \approx t\}$ is unifiable.

Next, we say that $E = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ is *trivial* iff $s_i = t_i$ for $i = 1, \ldots, n$. And for any substitution θ, we define $\overline{\theta}(E)$ as the system

$$\{\overline{\theta}(s_1) \approx \overline{\theta}(t_1), \ldots, \overline{\theta}(s_n) \approx \overline{\theta}(t_n)\}.$$

The following is immediate:

Lemma 9.16 *θ unifies E iff $\overline{\theta}(E)$ is trivial.*

A system E is said to be *in solved form* iff it is of the form

$$\{x_1 \approx t_1, \ldots, x_k \approx t_k\}$$

where the variables $x_1, \ldots, x_k$ are distinct and $x_i \notin Var(t_j)$ for any $i, j \in \{1, \ldots, k\}$. Clearly, such a system E determines a unique substitution

$$\theta_E = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$$

that is an idempotent most general unifier of E, as Lemma 9.18 states below. We will need one auxiliary result before we prove Lemma 9.18:

[4] Our discussion could also be cast in terms of simple finite sets, but the use of multisets avoids certain tedious complications.

[5] Recalling that a multiset S over a universe A is formally defined as a function $S : A \to N$, the union of two multisets S_1, S_2 over A can be defined as $\lambda\, x : A \,.\, S_1(x) + S_2(x)$.

Lemma 9.17 *If θ unifies $x \approx t$ then $\theta = \theta \circ \{x \mapsto t\}$.*

Proof: By supposition,

$$\theta(x) = \overline{\theta}(t). \tag{9.38}$$

Now pick any variable z. Either $z = x$ or not. If $z = x$ then

$$\begin{array}{rcll} \theta \circ \{x \mapsto t\}(z) & = & & \text{(definition of } \circ \text{)} \\ \overline{\theta}(\{x \mapsto t\}(z)) & = & & \text{(since } z = x\text{)} \\ \overline{\theta}(t) & = & & \text{(from 9.38)} \\ \theta(x) & = & \theta(z). & \end{array}$$

In contradistinction, if $z \neq x$ then $\theta \circ \{x \mapsto t\}(z) = \overline{\theta}(\{x \mapsto t\}(z)) = \overline{\theta}(z) = \theta(z)$. Thus $\theta(z) = \theta \circ \{x \mapsto t\}(z)$ for all z, which is to say $\theta = \theta \circ \{x \mapsto t\}$. ■

Lemma 9.18 *If the system $E = \{x_1 \approx t_1, \ldots, x_k \approx t_k\}$ is in solved form then the substitution $\theta_E = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ is an idempotent mgu of E.*

Proof: Idempotency follows from the supposition that $x_i \notin \mathit{Var}(t_j)$ for all i, j in $\{1, \ldots, k\}$. This also means that $\overline{\theta_E}(t_i) = t_i$ for $i = 1, \ldots, k$, hence $\overline{\theta_E}(t_i) = \theta_E(x_i)$ and θ_E unifies E. Now pick any $\sigma \in U(E)$. If a variable z is one of the x_i then, since σ unifies E, $\sigma(z) = \sigma(x_i) = \overline{\sigma}(t_i) = \overline{\sigma}(\theta_E(x_i)) = \overline{\sigma}(\theta_E(z))$, while if z is not one of the x_i then the equality

$$\sigma(z) = \overline{\sigma}(\theta_E(z)) \tag{9.39}$$

holds trivially, so 9.39 holds for all z, i.e., $\sigma = \sigma \circ \theta_E$. Accordingly, θ_E is more general than σ. ■

The HMM algorithm attempts to transform a given set of equations into solved form by repeated applications of the following rules:

- *Simplification*: $E, t \approx t \Longrightarrow E$;
- *Decomposition*: $E, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \Longrightarrow E, s_1 \approx t_1, \ldots, s_n \approx t_n$;
- *Transposition*: $E, t \approx x \Longrightarrow E, x \approx t$—provided that t is not a variable;
- *Application*: $E, x \approx t \Longrightarrow \overline{\{x \mapsto t\}}(E), x \approx t$—provided that $x \notin \mathit{Var}(t)$ and that x occurs in E.

For any two systems E and E', we write $E \Longrightarrow E'$ to signify that E' can be obtained from E by one of the rules.

The qualification in the transposition rule is needed to guarantee the termination of the transformation process. The same goes for the qualification that x must occur in E in the last rule (the second qualification of that rule, $x \notin \mathit{Var}(t)$, ensures that the process does not proceed in the presence of an equation $x \approx t$ with $x \in \mathit{Var}(t)$, since such an equation is not unifiable). Thus we see that these are not pure inference rules, in the sense that they have control information built into them, intended to ensure that they cannot be applied indefinitely. This will be made clear in Section 9.6.3, where it will be shown that these rules essentially perform *search* rather than *inference*.

Now the idea behind using these transformations as an algorithm for unifying two terms s and t is this: we start with the system $E_1 = \{s \approx t\}$ and keep applying rules (non-deterministically), building up a sequence $E_1 \Longrightarrow E_2 \Longrightarrow \cdots \Longrightarrow E_k$, until we finally arrive at a system of equations E_k to which no more rules can be applied. It is straightforward to prove termination (i.e., that it is impossible to continue applying rules ad infinitum), and that if s and t are indeed unifiable then the final system E_k will be in solved form, i.e., of the form $E_k = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, where the variables $x_1, \ldots, x_n$ are distinct and x_i does not occur in any t_j. Accordingly, the substitution $\theta_{E_k} = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent mgu of E_k. Further, we can show that if E_{i+1} is obtained from E_i by one of the rules—i.e., if $E_i \Longrightarrow E_{i+1}$—then $U(E_i) = U(E_{i+1})$, so that any substitution that unifies E_i also unifies E_{i+1} and vice versa. Thus it follows that θ_{E_k} is also an idempotent mgu of $E_{k-1}, E_{k-2}, \ldots, E_1$, and hence an idempotent mgu of s and t. On the other hand, if the final set of equations E_k is *not* in solved form then we may conclude that the initial terms s and t are not unifiable.

As an example, here is a series of transformations resulting in a most general unifier for the terms $f(x, g(z), b, z)$ and $f(a, y, b, h(x))$:

$$\begin{array}{lll}
1. & \{f(x,g(z),b,z) \approx f(a,y,b,h(x))\} & \Longrightarrow \quad \textit{Decompose} \\
2. & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} & \Longrightarrow \quad \textit{Apply } x \approx a \\
3. & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\} & \Longrightarrow \quad \textit{Apply } z \approx h(a) \\
4. & \{x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)\} & \Longrightarrow \quad \textit{Simplify} \\
5. & \{x \approx a, g(h(a)) \approx y, z \approx h(a)\} & \Longrightarrow \quad \textit{Transpose} \\
6. & \{x \approx a, y \approx g(h(a)), z \approx h(a)\} &
\end{array}$$

The system $\{x \approx a, y \approx g(h(a)), z \approx h(a)\}$ is in solved form, and thus the substitution

$$\{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$$

is an idempotent mgu of the given terms.

On the other hand, if we start with two terms that cannot be unified, say $f(a)$ and $f(b)$, or x and $g(x)$, we will end up with a system of equations that cannot be transformed any further but is not in solved form, and we can then conclude that unification is impossible.

9.6.2 Deductive formulation

We want to devise a calculus $\mathcal{U}$ for *proving* that a system of equations E is unifiable. We could use such a calculus to show that two given terms s and t can be unified by adducing a proof to the effect that the system $\{s \approx t\}$ is unifiable. Such a proof would start from axioms asserting that certain systems are evidently unifiable, and proceed by applying inference rules of the form "If $E_1, \ldots, E_n$ are unifiable then so is E". The HMM transformation rules are not appropriate for that purpose because they proceed in the reverse direction: they start from the equations whose unifiability we wish to establish and work their way back to sets of equations whose unifiability is apparent. In that sense, they are *analytic*, or "backwards" rules: they keep breaking up the original equations into progressively simpler components. By contrast, we want *synthetic* rules that will allow us to move in a forward manner: starting from simple elements, we must be able to build up the desired equations in a finite number of steps. In fact we will see shortly that the HMM algorithm is, in a very precise sense, a backwards proof-search algorithm for the deduction system we will set up below.

We must now decide exactly what form the judgments of our calculus will have. One simple choice is to work with judgments of the form $\vdash_U E$, asserting that the system E is unifiable. However, we will instead opt for more complex judgments, of the form $\vdash_U E : \theta$, asserting that the substitution θ is an idempotent most general unifier of E. The advantage of such a judgment is that it conveys more information that the mere fact that E is unifiable; it includes a substitution θ that actually unifies E. In addition, the judgment guarantees that θ is idempotent and most general. This design choice will enable us to use our logic for computational purposes, namely, for computing unifiers. More precisely, it will enable us, when we come to formulate our logic as a $\lambda\mu$ system, to write a method that takes two terms s and t as input and—provided that s and t are unifiable—returns a theorem of the form $\{s \approx t\} : \theta$. This theorem does not only tell us that the terms s and t are unifiable, it also produces an idempotent mgu for them. By now the reader should be able to guess the benefit of computing unifiers with such a method (rather than with a function): guaranteed correctness. If we do obtain a theorem $\{s \approx t\} : \theta$, we can be assured that θ is indeed an idempotent mgu of s and t.

The logic comprises one axiom and four unary rules, shown in Figure 9.5. The

$$\frac{}{\vdash_U \{x_1 \approx t_1, \ldots, x_k \approx t_k\} : \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}} \;[\texttt{Solved-Form}]$$

provided $\{x_1 \approx t_1, \ldots, x_k \approx t_k\}$ is in solved form

$$\frac{\vdash_U E : \theta}{\vdash_U E, t \approx t : \theta} \;[\texttt{Reflexivity}]$$

$$\frac{\vdash_U E, s \approx t : \theta}{\vdash_U E, t \approx s : \theta} \;[\texttt{Symmetry}]$$

$$\frac{\vdash_U E, s_1 \approx t_1, \ldots, s_n \approx t_n : \theta}{\vdash_U E, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) : \theta} \;[\texttt{Congruence}]$$

$$\frac{\vdash_U E, x \approx t : \theta}{\vdash_U E', x \approx t : \theta} \;[\texttt{Abstraction}]$$

provided $\overline{\{x \mapsto t\}}(E') = E$.

Figure 9.5: A logic for deducing idempotent most general unifiers.

axiom [`Solved-Form`] asserts that every set of equations $E = \{x_1 \approx t_1, \ldots, x_k \approx t_k\}$ in solved form is unifiable, and that $\theta_E = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ is an idempotent mgu of E, which is clearly true by Lemma 9.18. The rules [`Reflexivity`], [`Symmetry`], and [`Congruence`] are self-explanatory, and their soundness should be clear; we will formally prove it shortly. Observe that if we read the rules in a forward manner then, in relation to the HMM transformations, reflexivity can be viewed as the inverse of simplification, symmetry as the inverse of transposition, and congruence as the inverse of decomposition. We will also see that [`Abstraction`] is the inverse of application. Also notice that these are pure inference rules, in the sense that no control information is embedded in them. Restrictions such as found in the transposition rule of the HMM system will instead be relegated to the control structure of a method that automates the logic $\mathcal{U}$, keeping the logic itself cleaner.

The rule [`Abstraction`] is a bit trickier. The key is the proviso $\overline{\{x \mapsto t\}}(E') = E$. This means that the equations in E' are abstractions of the equations in E, obtainable from the latter by replacing certain occurences of t by x. Alternatively, the equations in E are *instances* of the equations in E', obtained from the latter by applying the substitution $\{x \mapsto t\}$. (Indeed, that is how the rule would be applied in a backwards fashion, and we will see that this is precisely the sense in which the HMM application rule is the inverse of abstraction.) Accordingly, the equations of E' are *more general* than those of E; and this is why the rule is called "abstraction": it takes us from the specific to the general.

Let us illustrate with our earlier example. We wish to show that $f(x, g(z), b, z)$ and $f(a, y, b, h(x))$ are unifiable, or more precisely, that the substitution

$$\theta = \{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$$

is an idempotent mgu of these two terms. The following deduction proves this:

1.	$\{x \approx a, y \approx g(h(a)), z \approx h(a)\} : \theta$	[`Solved-Form`]
2.	$\{x \approx a, g(h(a)) \approx y, z \approx h(a)\} : \theta$	1, [`Symmetry`]
3.	$\{x \approx a, g(z) \approx y, z \approx h(a)\} : \theta$	2, [`Abstraction`] on $z \approx h(a)$
4.	$\{x \approx a, g(z) \approx y, z \approx h(x)\} : \theta$	3, [`Abstraction`] on $x \approx a$
5.	$\{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} : \theta$	4, [`Reflexivity`]
6.	$\{f(x, g(z), b, z) \approx f(a, y, b, h(x))\} : \theta$	5, [`Congruence`]

Note that the only rule that creates—or in any way affects—the substitution θ of a judgment $E : \theta$ is the axiom [`Solved-Form`]. All the other rules simply pass along the substition of the premise unchanged. Thus a substitution is created only once, for a system in solved form, and from that point on it is carried along from system to system via the various rules, until it is finally attached to the desired system.

We will now prove that the rules of our logic are sound. We will need the following two lemmas:

Lemma 9.19 *If* $\overline{\{x \mapsto t\}}(E') = E$ *then* $U(E, x \approx t) = U(E', x \approx t)$.

Proof: In one direction, suppose that θ unifies $E, x \approx t$.[6] Then, from Lemma 9.17, $\theta = \theta \circ \{x \mapsto t\}$, hence

$$\begin{aligned} \overline{\theta}(E') = \overline{\theta \circ \{x \mapsto t\}}(E') &= \quad \text{(Lemma A.1)} \\ \overline{\theta}(\overline{\{x \mapsto t\}}(E')) &= \quad \text{(supposition)} \\ \overline{\theta}(E). & \end{aligned}$$

But θ unifies E, so from Lemma 9.16, $\overline{\theta}(E)$ is trivial, and by the foregoing equality,

[6] Recall that we write $E, s \approx t$ for the multiset union of E and $\{s \approx t\}$.

$\overline{\theta}(E')$ is trivial. Thus θ unifies E', and a fortiori, $E', x \approx t$. A symmetrical argument will establish the converse inclusion. ■

Lemma 9.20 *If $U(E_1) = U(E_2)$ then θ is a mgu of E_1 iff it is a mgu of E_2.*

Proof: Let θ be a mgu of E_1. Then θ unifies E_2, and for any $\sigma \in U(E_2)$ we also have $\sigma \in U(E_1)$, hence $\sigma \leq \theta$. Thus θ is a mgu of E_2. The converse implication is shown likewise. ■

We will say that a judgment $E : \theta$ *holds* iff θ is an idempotent mgu of E. Accordingly, a rule of the form "From $E_1 : \theta_1, \ldots, E_n : \theta_n$ infer $E : \theta$" will be considered sound iff the conclusion $E : \theta$ holds whenever each premise $E_i : \theta_i$ holds, $i = 1, \ldots, n$. We can now prove:

Theorem 9.21 *The rules of $\mathcal{U}$ are sound.*

Proof: The soundness of the axiom [`Solved-Form`] follows from Lemma 9.18. The other four rules are unary, i.e., of the form "From $E_1 : \theta$ infer $E_2 : \theta$", and we will show that in all four cases we have $U(E_1) = U(E_2)$; soundness will then follow from Lemma 9.20. For reflexivity, symmetry, and congruence we need to show, respectively, the identities $U(E) = U(E, t \approx t)$, $U(E, s \approx t) = U(E, t \approx s)$, and

$$U(E, s_1 \approx t_1, \ldots, s_n \approx t_n) = U(E, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)),$$

all three of which are immediate. In the case of abstraction the desired identity follows from Lemma 9.19. ■

9.6.3 The unification calculus as a $\lambda\mu$ system

In this section we will cast the logic $\mathcal{U}$ as a $\lambda\mu$ system, *Unif*. As constants we take:

- every element of $\mathcal{F} \cup \mathcal{V}$;
- every term over $\mathcal{F}$ and $\mathcal{V}$;
- all ordered pairs of such terms, to serve as equations;
- all systems of equations (finite multisets thereof);
- all substitutions; and

$$\emptyset \vdash \texttt{!solved}\ E \rightsquigarrow E : \theta_E$$
provided E is in solved form

$$\{E, x \approx t : \theta\} \vdash \texttt{!abstract}\ E, x \approx t : \theta\ \ x \approx t\ \ E' \rightsquigarrow E', x \approx t : \theta$$
provided $\overline{\{x \mapsto t\}}(E') = E$

$$\{E : \theta\} \vdash \texttt{!ref}\ E : \theta\ \ t \approx t \rightsquigarrow E, t \approx t : \theta$$

$$\{E, s \approx t : \theta\} \vdash \texttt{!sym}\ E, s \approx t : \theta\ \ t \approx s \rightsquigarrow E, t \approx s : \theta$$

$$\{E : \theta\} \vdash \texttt{!cong}\ E : \theta\ \ f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) \rightsquigarrow$$
$$E', f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n) : \theta$$
whenever E is of the form $E', s_1 \approx t_1, \ldots, s_n \approx t_n$.

Figure 9.6: δ-rules for the primitive methods of *Unif*.

- all ordered pairs $\langle E, \theta \rangle$ consisting of a system E and a substitution θ.

The latter will be the *sentences* of the system. For increased readability, a sentence $\langle E, \theta \rangle$ is written as $E : \theta$. We will say that such a sentence *holds* iff θ is an idempotent most general unifier of E.

We define an entailment relation $\models$ as follows: $\beta \models E : \theta$ iff $E : \theta$ holds whenever every member of β holds (keep in mind that an assumption base β is a set of sentences, so in this system β will be a set of ordered pairs $\langle E, \theta \rangle$). It is readily verified that $\models$ is a Tarskian relation.

The primitive methods of *Unif* will be the five constants solved, abstract, ref, sym, and cong. Their semantics are given by the δ-evaluation axioms shown in Figure 9.6. The reader should convince himself that all five methods satisfy the requirements **PM1** and **PM2** laid down in Section 8.2.

We can now present our earlier proof that $\{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$ is an idempotent mgu of the terms $f(x, g(z), b, z)$ and $f(a, y, b, h(x))$ as a $\lambda\mu$ deduction:

$$\begin{aligned}
\textbf{dlet}\ \ & S_1 = \texttt{!solved}\ \{x \approx a, y \approx g(h(a)), z \approx h(a)\} \\
& S_2 = \texttt{!sym}\ S_1\ \ g(h(a)) \approx y \\
& S_3 = \texttt{!abstract}\ S_2\ \ z \approx h(a)\ \ \{x \approx a, g(z) \approx y\} \\
& S_4 = \texttt{!abstract}\ S_3\ \ x \approx a\ \ \{g(z) \approx y, z \approx h(x)\} \\
& S_5 = \texttt{!ref}\ S_4\ \ b \approx b
\end{aligned}$$

in
  $!\mathsf{cong}\ S_5\ \ f(x, g(z), b, z) \approx f(a, y, b, h(x))$

Note that this proof is more succinct than the informal one given in page 357.

We may also express certain parts of our choice in conclusion-annotated style, resulting, for instance, in:

dlet $E = \{x \approx a, y \approx g(h(a)), z \approx h(a)\}$
  $\theta = \{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$
  $S_1 = E : \theta$ **by** $!\mathsf{solved}\ \ E$
  $S_2 = \{x \approx a, g(h(a)) \approx y, z \approx h(a)\} : \theta$ **by** $!\mathsf{sym}\ \ S_1\ \ g(h(a)) \approx y$
  $S_3 = \{x \approx a, g(z) \approx y, z \approx h(a)\} : \theta$ **by** $!\mathsf{abstract}\ \ S_2\ \ z \approx h(a)\ \ \{x \approx a, g(z) \approx y\}$
  $S_4 = \{x \approx a, g(z) \approx y, z \approx h(x)\} : \theta$ **by** $!\mathsf{abstract}\ \ S_3\ \ x \approx a\ \ \{g(z) \approx y, z \approx h(x)\}$
  $S_5 = \{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} : \theta$ **by** $!\mathsf{ref}\ \ S_4\ \ b \approx b$
in
  $\{f(x, g(z), b, z) \approx f(a, y, b, h(x))\} : \theta$ **by** $!\mathsf{cong}\ \ S_5\ \ f(x, g(z), b, z) \approx f(a, y, b, h(x))$

As another example, here is a proof showing that $\{x \mapsto a, y \mapsto h(a), z \mapsto b\}$ is an idempotent mgu of the terms $f(x, g(y, z))$ and $f(a, g(h(x), b))$:

dlet $S_1 = !\mathsf{solved}\ \ \{x \approx a, y \approx h(a), z \approx b\}$
  $S_2 = !\mathsf{abstract}\ \ S_1\ \ x \approx a\ \ \{y \approx h(x), z \approx b\}$
  $S_3 = !\mathsf{cong}\ \ S_2\ \ g(y, z) \approx g(h(x), b)$
in
  $!\mathsf{cong}\ \ S_3\ \ f(x, g(y, z)) \approx f(a, g(h(x), b))$

or in conclusion-annotated style:

dlet $E = \{x \approx a, y \approx h(a), z \approx b\}$
  $\theta = \{x \mapsto a, y \mapsto h(a), z \mapsto b\}$
  $S_1 = E : \theta$ **by** $!\mathsf{solved}\ \ E$
  $S_2 = \{x \approx a, y \approx h(x), z \approx b\} : \theta$ **by** $!\mathsf{abstract}\ \ S_1\ \ x \approx a\ \ \{y \approx h(x), z \approx b\}$
  $S_3 = \{x \approx a, g(y, z) \approx g(h(x), b)\} : \theta$ **by** $!\mathsf{cong}\ \ S_2\ \ g(y, z) \approx g(h(x), b)$
in
  $\{f(x, g(y, z)) \approx f(a, g(h(x), b))\} : \theta$ **by** $!\mathsf{cong}\ \ S_3\ \ f(x, g(y, z)) \approx f(a, g(h(x), b))$

We now turn to soundness and completeness.

Theorem 9.22 (Soundness) *Unif is sound, i.e., if $\beta \vdash D \rightsquigarrow E : \theta$ then $\beta \models E : \theta$. In particular, if $\emptyset \vdash D \rightsquigarrow E : \theta$ then E is unifiable, and θ is an idempotent mgu of it.*

$unify = \mu\, E\,.$
;; Case 1:
  dcond (E in solved form?)
    $!\mathsf{solved}\;\; E$
;; Case 2:
  dcond (E of the form $E', x \approx t$ where $x \notin \mathit{Var}(t)$ and x occurs in E'?)
    dlet $S = !\,unify\;\; \overline{\{x \mapsto t\}}(E'), x \approx t$
    in
      $!\mathsf{abstract}\;\; S\;\; x \approx t\;\; E'$
;; Case 3:
  dcond (E of the form $E', t \approx x$ where t is not a variable?)
    dlet $S = !\,unify\;\; E', x \approx t$
    in
      $!\mathsf{sym}\;\; S\;\; t \approx x$
;; Case 4:
  dcond (E of the form $E', t \approx t$?)
    dlet $S = !\,unify\;\; E'$
    in
      $!\mathsf{ref}\;\; S\;\; t \approx t$
;; Case 5:
  dcond (E of the form $E', f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)$?)
    dlet $S = !\,unify\;\; E', s_1 \approx t_1, \ldots, s_n \approx t_n$
    in
      $!\mathsf{cong}\;\; S\;\; f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)$

Figure 9.7: A unification method.

Proof: $\models$ is Tarskian, so by Theorem 8.6 we only need to check that $\models$ includes every primitive method. We have already done this in the proof of Theorem 9.21. ∎

The usual question of tautological completeness—whether we can prove every sentence that holds—is not of much interest in the present context, because it assumes the following form: suppose $E : \theta$ holds, i.e., that θ is an idempotent mgu of E; can we prove this? The reason why this is not particularly interesting is that the unifier θ is already given as part of the input. For our purposes, a more practical question is this: suppose that E is unifiable; can we constructively prove the proposition

$$(\exists\, \theta)\, E : \theta$$

i.e., can we constructively prove the existence of an idempotent mgu for E? More precisely: can we prove the sentence $E : \theta$ for some θ that is appropriately constructed on the basis of the given E? If we can do this whenever E is unifiable, and fail whenever it is not, then we will have a unification "algorithm" that backs its results by proving that they are correct.

The method *unify* shown in Figure 9.7 answers this question affirmatively. It takes an arbitrary system E as input and, as we will show shortly, it derives a theorem of the form $E : \theta$ iff E is unifiable. Implementing the predicates of the five **dcond** clauses in the body of the method is a straightforward exercise that is peripheral to the main task, and therefore, to save space, we have expressed the predicates in plain English. This introduces some non-determinism: for instance, in case 4, E might contain several equations of the form $t = t$, and there will then be several possible choices for E' and t; and likewise for all other cases, with the exception of the first. However, the non-determinism will be eliminated once the English descriptions are replaced by $\lambda\mu$ code. The interested reader will readily achieve this replacement after settling on a few primitive functions for manipulating multisets.

We will now study the behavior of *unify* in detail, starting with the following simple observation:

Lemma 9.23 *If* $(\mathbf{!}\,\mathit{unify}\ E)$ *terminates successfully, in any assumption base, then the theorem produced is of the form* $E : \theta$*.*

Proof: By inspection of the five possible cases and the definitions of the primitive methods and multiset union. ■

For the next result we need to define, for any term t, the quantity $FS(t)$, denoting the number of function symbols occuring in t. More precisely, we set

$$\begin{array}{rcl} FS(c) & = & 1 \\ FS(x) & = & 0 \\ FS(f(t_1, \ldots, t_n)) & = & 1 + FS(t_1) + \cdots + FS(t_n). \end{array}$$

Further, recall that $SZ(t)$ denotes the size of a term t, i.e., the number of function symbol occurences plus the number of variable occurences.

Theorem 9.24 *The method unify always terminates.*

Proof: Let us say that a variable x occuring in a system of equations E is *solved in* E iff E is of the form $E', x \approx t$, where $x \notin \mathit{Var}(t)$ and x does not occur in E'. In other words, x is solved in E iff x occurs in exactly one equation in E, and that equation is of the form $x \approx t$, with $x \notin \mathit{Var}(t)$. Otherwise we will say that x is *unsolved in* E. For any $E =$

$\{s_1 \approx t_1, \ldots, s_n \approx t_n\}$, let $UV(E)$, $LS(E)$, and $LFS(E)$ denote the number of unsolved variables in E, $SZ(s_1)+\cdots+SZ(s_n)$, and $FS(s_1)+\cdots+FS(s_n)$, respectively. We claim that if a method call (**!** *unify* E_1)—placed in the context of an arbitrary assumption base—results in a recursive call (**!** *unify* E_2) then the triple $\langle UV(E_2), LS(E_2), LFS(E_2)\rangle$ is lexicographically smaller than $\langle UV(E_1), LS(E_1), LFS(E_1)\rangle$; i.e., either $UV(E_2) < UV(E_1)$; or $UV(E_2) = UV(E_1)$ and $LS(E_2) < LS(E_1)$; or else $UV(E_2) = UV(E_1)$, $LS(E_2) = LS(E_1)$, and $LFS(E_2) < LFS(E_1)$. This means that we cannot have an infinitely long chain of recursive calls, since a lexicographic extension of a well-founded relation is itself well-founded.

To verify our claim one must check each of the four possible recursive calls in the body of *unify* (cases 2—5). In the recursive call of case 2 the number of unsolved variables strictly decreases, so the claim holds. The recursion of case 3 (sym) does not increase the number of unsolved variables (since t is not a variable) or the quantity LS (since the size of t must be at least 1), and decreases the quantity LFS (because t is not a variable and gets transferred to the right side). In case 4, UV does not increase while LS decreases; and the same holds for the recursion of case 5. ■

Lemma 9.25 *If E is unifiable then at least one the five cases distinguished in the body of unify must hold.*

Proof: We will show that if cases 2—5 do not hold then E must be in solved form, so that case 1 holds. We begin by observing that any given equation must be of exactly one of the following three kinds:

(i) $x \approx t$; or

(ii) $f(s_1, \ldots, s_n) \approx x$; or

(iii) $f(s_1, \ldots, s_n) \approx g(t_1, \ldots, t_m)$

where, for convenience, a constant symbol is represented as $f()$. Now pick any equation $s \approx t$ in E. The equation cannot be of the form (ii), because then case 3 would apply. Suppose, next, that $s \approx t$ is of the form (iii). Then we must have $f = g$, since $f \neq g$ would imply that E is not unifiable, contrary to our supposition. Now either $n = 0$, i.e., f is a constant symbol, or $n > 0$. But because $f = g$, if $n = 0$ then case 4 would apply, while if $n > 0$ case 5 would apply. Since in either case we get a contradiction, we conclude that the equation in question cannot be as shown in (iii) either; thus it must be of the form (i).

Furthermore, since $x \neq t$, we must have $x \notin Var(t)$, for otherwise E would not be unifiable. But then it must also be the case that x does not occur anywhere else in E, for otherwise case 2 would obtain. We have thus shown that every equation in E is of the form $x \approx t$, where x occurs neither in t nor anywhere else in E. Accordingly, E is in solved form. ■

Theorem 9.26 *If E is unifiable then the method call* (**!** *unify* E) *will produce—in any assumption base—a sentence of the form $E : \theta$, where, by soundness, θ is an idempotent mgu of E.*

Proof: By the previous result, one of the five cases in the body of *unify* must obtain. In the first case the result is immediate, while in the remaining four cases we proceed by well-founded induction on $\langle UV(E), LS(E), LFS(E)\rangle$. The discussion in the proof of Theorem 9.21 shows that the argument to each of the four recursive calls is also unifiable, hence the inductive hypothesis applies, and the result in each case follows directly from induction and the δ-evaluation axiom for the corresponding primitive method. ■

These results clarify the sense in which we can use this logic for computational purposes. Supposing that we wish to determine whether two terms s and t are unifiable, and if so, to find an idempotent mgu for them, we can invoke *unify* with the system $\{s \approx t\}$ as input. If s and t are indeed unifiable then the above result guarantees that we will obtain a theorem of the form $\{s \approx t\} : \theta$, where θ is an idempotent mgu for s and t. On the other hand, if s and t are not unifiable then the method will fail, since it always terminates and yet could not terminate successfully for then the theorem produced would be of the form $E : \theta$ (Lemma 9.23), implying (by soundness) that s and t are unifiable, contrary to our supposition.

Let us illustrate with the terms $f(x, g(z), b, z)$ and $f(a, y, b, h(x))$. Figure 9.8 shows the flow of control during the evaluation of the call

$$\mathbf{!}\,\mathit{unify}\ \ \{f(x, g(z), b, z) \approx f(a, y, b, h(x))\} \tag{9.40}$$

in a counter-clockwise direction. (To save space, we have written the substitution $\{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$ as θ.) Observe that the recursive calls that are made during the evaluation, depicted on the left side of the diagram, correspond precisely to HMM transformations. This shows that HMM transformations embody control strategy—in particular, proof-search strategy—rather than inference rules. The actual inferences that make up the discovered proof are wide asunder: they consist of the primitive method applications shown on the right side of the picture, beginning at the bottom with the application of the axiom `solved` and culminating in the upper right-hand corner with the application of `cong`. The HMM transformations are only useful in *searching* for the proof in a backwards manner; once we have found the apprjudge

$$
\begin{tikzcd}
\mathbf{!}\,\mathit{unify}\ \{f(x, g(z), b, z) \approx f(a, y, b, h(x))\} \arrow[d, "\text{Case 5 (decompose)}"] \arrow[r, phantom, "\xleftrightarrow{\text{Inverses}}"] & \{f(x, g(z), b, z) \approx f(a, y, b, h(x))\} : \theta \\
\mathbf{!}\,\mathit{unify}\ \{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} \arrow[d, "\text{Case 2 (apply)}"] & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\} : \theta \arrow[u, "\mathbf{!}\mathsf{cong}"'] \\
\mathbf{!}\,\mathit{unify}\ \{x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\} \arrow[d, "\text{Case 2 (apply)}"] & \{x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathbf{!}\mathsf{abstract}"'] \\
\mathbf{!}\,\mathit{unify}\ \{x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)\} \arrow[d, "\text{Case 3 (transpose)}"] & \{x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathbf{!}\mathsf{abstract}"'] \\
\mathbf{!}\,\mathit{unify}\ \{x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)\} \arrow[d, "\text{Case 4 (simplify)}"] & \{x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)\} : \theta \arrow[u, "\mathbf{!}\mathsf{sym}"'] \\
\mathbf{!}\,\mathit{unify}\ \{x \approx a, y \approx g(h(a)), z \approx h(a)\} \arrow[r, "\mathbf{!}\mathsf{solved}"'] & \{x \approx a, y \approx g(h(a)), z \approx h(a)\} : \theta \arrow[u, "\mathbf{!}\mathsf{ref}"']
\end{tikzcd}
$$

Figure 9.8: Control flow, shown counter-clockwise, for the method application 9.40, where $\theta = \{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$. Note that the recursive calls correspond precisely to HMM transformations.

starting point (axiom), we put the proof together in a forward manner with the proper inference rules.

The diagram does not merely show that the HMM rules are applied in the opposite direction from the primitive methods, but spells out the exact inverse relationship between the four HMM transformations and the four primitive methods of *Unif* (the axiom solved being neutral). Specifically, we can see that for every HMM transformation there is exactly one primitive method that reverses ("undoes") the transformation: congruence undoes decomposition, symmetry undoes transposition, reflexivity undoes simplification, and abstraction undoes application.

The picture also reinforces our earlier point that the only method which has any sort of impact on the final substitution θ of a theorem $E : \theta$ is the axiom `solved`. All the other methods simply pass along their given substition unchanged. So the strategy of *unify*, as the figure illustrates, is to transform the given system into solved form, create a substitution for that solved system, and then *prove* that this substitution is also an idempotent mgu for the original system. That half of the process, depicted on the right side of the picture, is what differentiates the method from the HMM algorithm, and the source of the correctness guarantee.

Finally, returning to the issue of tautological completeness, we remark that if we consider substitution identity modulo renaming (so that two substitutions are regarded as identical iff each can be obtained from the other via composition with a renaming), as is customarily done, then our logic is complete: the method *unify* can be used to derive all tautologies, since mgus are unique up to renaming. Specifically, if a sentence $E : \theta$ is valid (holds), then (**!***unify* E) will result in the theorem $E : \theta'$, where θ' will be identical to θ modulo renaming. However, *unify* cannot be used as a solution to the validity problem (and hence as a decision procedure for theoremhood), because if E *is* unifiable but θ is *not* a unifier of it then we will fail to reject the given sentence $E : \theta$, since (**!***unify* E) will successfully produce a theorem $E : \theta'$ (with θ' distinct from θ, of course). To solve the validity problem with a method, then, we would need to introduce additional inference rules specifying when a substitution is more general than another. We will not pursue this theme here, as it is tangential to the subject of unification. We summarize:

Theorem 9.27 *The $\lambda\mu$ system Unif is sound, consistent, and complete.*

9.7 Natural deduction in the $\lambda\mu$-calculus, predicate case

9.7.1 Definition

Fix a logic vocabulary $(\mathbf{\Omega}, \mathcal{V})$, consisting of a signature $\mathbf{\Omega} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ and a countably infinite set of variables $\mathcal{V}$ (see Section 6.1). We will assume that there is a given well-order $\prec_{\mathcal{V}}$ on $\mathcal{V}$, such that the unique $\prec_{\mathcal{V}}$-successor of any given $x \in \mathcal{V}$ can be mechanically computed (recall that x is used as a metavariable ranging over $\mathcal{V}$). Free and bound occurences of variables, etc., are defined as usual. Alphabetically equivalent formulas (Figure 6.1) are considered identical.

The $\lambda\mu$ system that we define in this section, $\lambda\mu$–$\mathcal{CND}_1$, will be parameterized over the vocabulary $(\mathbf{\Omega}, \mathcal{V})$. The primitive values of $\lambda\mu$–$\mathcal{CND}_1$ include all terms $t \in \mathbf{Terms}(\mathbf{\Omega}, \mathcal{V})$ and formulas $F \in \mathbf{Form}(\mathbf{\Omega}, \mathcal{V})$. The latter will be the system's sentences.

For each function symbol $f \in \mathcal{F}$ of arity $n > 0$ we include a primitive $\lambda\mu$ function f of the same arity and with the following δ-rules:

$$\beta \vdash \mathbf{app}(\mathsf{f}, t_1, \ldots, t_n) \rightsquigarrow f(t_1, \ldots, t_n).$$

Likewise, for every relation symbol $R \in \mathcal{R}$ of arity n we have a primitive function R with semantics:

$$\beta \vdash \mathbf{app}(\mathsf{R}, t_1, \ldots, t_n) \rightsquigarrow R(t_1, \ldots, t_n).$$

We also include five primitive functions $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ for building formulas:

$$\begin{gathered}
\beta \vdash \mathbf{app}(\neg, F) \rightsquigarrow \neg F \\
\beta \vdash \mathbf{app}(\wedge, F_1, F_2) \rightsquigarrow F_1 \wedge F_2 \\
\beta \vdash \mathbf{app}(\vee, F_1, F_2) \rightsquigarrow F_1 \vee F_2 \\
\beta \vdash \mathbf{app}(\Rightarrow, F_1, F_2) \rightsquigarrow F_1 \Rightarrow F_2 \\
\beta \vdash \mathbf{app}(\Leftrightarrow, F_1, F_2) \rightsquigarrow F_1 \Leftrightarrow F_2
\end{gathered}$$

and two binary primitive functions $\forall$ and $\exists$ for building quantified formulas:

$$\begin{gathered}
\beta \vdash \mathbf{app}(\forall, x, F) \rightsquigarrow (\forall\, x)\, F \\
\beta \vdash \mathbf{app}(\exists, x, F) \rightsquigarrow (\exists\, x)\, F
\end{gathered}$$

We refer to a primitive function f (for $f \in \mathcal{F}$) as a *term constructor*. The primitive functions in $\{\mathsf{R} \mid R \in \mathcal{R}\} \cup \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall, \exists\}$ are called *formula constructors*. More specifically, the elements of $\{\mathsf{R} \mid R \in \mathcal{R}\}$ are called *atomic* formula constructors; those in $\{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$ are called *propositional* formula constructors; while $\forall$ and $\exists$ are called *quantified* formula constructors.

In addition, we introduce a ternary primitive function sub (for substitution), with the semantics

$$\beta \vdash \mathbf{app}(\mathsf{sub}, t, x, F) \rightsquigarrow \{x \mapsto t\}\, F.$$

Thus an application $\mathbf{app}(\mathsf{sub}, t, x, F)$ should be read as "substitute t for all free occurences of x in F", and produces the sentence $\{x \mapsto t\}\, F$ (with a possible α-renaming to avoid variable capture). Instead of the usual abbreviation ($\mathsf{sub}\ M_1\ M_2\ N$), we will often write $\{M_2 \mapsto M_1\}\, N$ as a shorthand for $\mathbf{app}(\mathsf{sub}, M_1, M_2, N)$. The reader will verify that all of the foregoing primitive functions satisfy **PF1** and **PF2**.

The primitive methods of $\lambda\mu$–$\mathcal{CND}_1$ include the propositional methods T-axiom, F-axiom, mp, mt, dn, both, left-and, right-and, cd, left-either, right-either, equiv, left-iff, right-iff, and absurd. Their semantics are just as in $\lambda\mu$–$\mathcal{CND}_0$ (Figure 9.2), only now instead of propositions P, Q, etc., we are dealing with formulas F, G, etc. Thus, for instance, the δ-rules for mp are given by the schema

$$\{F \Rightarrow G, F\} \vdash \mathbf{dapp}(\mathsf{mp}, F \Rightarrow G, F) \rightsquigarrow G.$$

In addition, there are three new primitive methods: uspec, egen, and qs-axiom, with the following semantics:

$$\begin{gathered}
\{(\forall\, x)\, F\} \vdash \mathbf{dapp}(\mathsf{uspec}, (\forall\, x)\, F, t) \rightsquigarrow \{x \mapsto t\}\, F \\
\{\{x \mapsto t\}\, F\} \vdash \mathbf{dapp}(\mathsf{egen}, (\exists\, x)\, F, t) \rightsquigarrow (\exists\, x)\, F
\end{gathered}$$

$$\emptyset \vdash \mathbf{dapp}(\textsf{qs-axiom}, (\forall\, x)\, [F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G]) \rightsquigarrow$$
$$(\forall\, x)\, [F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G]$$
provided x does not occur free in G.

The reader will verify that all of the aforementioned primitive methods adhere to the provisos **PM1** and **PM2**.

Lastly, $\lambda\mu$–$\mathcal{CND}_1$ has two special deductive forms, $\mathbf{assume}(M, D)$ and

$$\mathbf{ugen}(E, D)$$

written, respectively, as **assume** M **in** D and **generalize-over** E **in** D. Their semantics are given by the following rules:

$$\frac{\beta \vdash M \rightsquigarrow^* F \quad \beta \cup \{F\} \vdash D \rightsquigarrow G}{\beta \vdash \mathbf{assume}(M, D) \rightsquigarrow F \Rightarrow G} \quad [\mathbf{assume}]$$

$$\frac{\beta \vdash E \rightsquigarrow^* x \quad \beta \vdash D \rightsquigarrow F}{\beta \vdash \mathbf{ugen}(E, D) \rightsquigarrow (\forall\, x)\, F} \quad [\mathbf{ugen}]$$
provided x does not occur free in β.

It is clear that both forms are as required in Section 8.2.3.

Universal generalization, like hypothetical reasoning, is another mode of inference that cannot be captured by a simple primitive method (the reader is invited to try to formulate such a primitive method); a special deductive form along the lines of $\mathbf{ugen}(E, D)$ is necessary. The reason is the need to ensure that the formula which we wish to generalize has been deductively derived from an assumption base that does not contain any free occurences of the relevant eigenvariable. Thus, for example, a primitive method ugen with semantics such as

$$\beta \cup \{F\} \vdash \mathbf{dapp}(\textsf{ugen}, x, F) \rightsquigarrow (\forall\, x)\, F$$

would be unsound, even if we made the provision $x \notin FV(\beta)$ (take $\beta = \emptyset$ and $F = \mathit{Even}(x)$).

Finally, assuming the availability of lists (see page 346 for how lists can be added to any $\lambda\mu$ system), we introduce a unary primitive function fresh-var with the following semantics:

$$\beta \vdash \mathbf{app}(\textsf{fresh-var}, [x_1, \ldots, x_n]) \rightsquigarrow x$$
where x is the $\prec_{\mathcal{V}}$–least variable that does not occur in $FV(\beta) \cup \{x_1, \ldots, x_n\}$

The reader should have little trouble verifying that the result of fresh-var is uniquely determined on the basis of the context β and the argument list $[x_1, \ldots, x_n]$, and can be mechanically computed from them. Accordingly, fresh-var satisfies **PF1** and **PF2**. Observe, however, that fresh-var is context-dependent, unlike the preceding primitive functions.

9.7.2 Metatheory

The entailment relation $\models$ from sets of formulas to formulas is defined as usual (see Section 6.5). We are ready to demonstrate soundness:

Theorem 9.28 *$\lambda\mu$–$\mathcal{CND}_1$ is sound, i.e., if $\Phi \vdash F$ then $\Phi \models F$.*

Proof: By Theorem 8.7, all we have to do is prove that the primitive methods are included in $\models$, and that the latter is preserved by the two special deductive forms **assume** and **ugen**. For the propositional primitive methods, the proof is as in $\lambda\mu$–$\mathcal{CND}_0$. From the three new primitive methods, the soundness of uspec and egen is shown just as in the case of first-order $\mathcal{CND}$ (see the proofs of Lemma 6.81 and Lemma 6.82, respectively). Thus we only need to show the soundness of qs-axiom. To that end, pick an arbitrary $\mathbf{\Omega}$-structure $\mathcal{D}$ and valuation $\rho : \mathcal{V} \to D$, and suppose that

$$\mathcal{D} \models_\rho (\forall\, x)\, [F \Rightarrow G] \tag{9.41}$$

and

$$\mathcal{D} \models_\rho (\exists\, x)\, F. \tag{9.42}$$

Now 9.42 means that there is a $d \in D$ such that

$$\mathcal{D} \models_{\rho[x \,\mapsto\, d]} F \tag{9.43}$$

and by 9.41 we get

$$\mathcal{D} \models_{\rho[x \,\mapsto\, d]} F \Rightarrow G, \tag{9.44}$$

so from 9.43 and 9.44,

$$\mathcal{D} \models_{\rho[x \,\mapsto\, d]} G. \tag{9.45}$$

But $x \notin FV(G)$, hence ρ and $\rho[x \mapsto d]$ agree on $FV(G)$ and the Coincidence Lemma (Lemma 6.73) gives $\mathcal{D} \models_\rho G$. We have thus shown that if $x \notin FV(G)$ then

$$\mathcal{D} \models_\rho (\forall\, x)\, [F \Rightarrow G] \Rightarrow [(\exists\, x)\, F \Rightarrow G]$$

for arbitrary $\mathcal{D}$ and ρ.

Finally, we have to show that **assume** and **ugen** preserve $\models$. For **assume** this is done just as in the propositional case. For **ugen**, we need to prove that, for any β, $\beta \models (\forall\, x)\, F$ whenever $\beta \models F$ and $x \notin FV(\beta)$. To that end, suppose that $\mathcal{D} \models_\rho \beta$, so that $\beta \models F$ gives $\mathcal{D} \models_\rho F$ (for arbitrary $\mathcal{D}$ and ρ). Pick any $d \in D$. Since $x \notin FV(\beta)$, the valuations ρ and $\rho[x \mapsto d]$ agree on the free variables of β, hence $\mathcal{D} \models_{\rho[x \,\mapsto\, d]} \beta$, and thus $\beta \models F$ implies $\mathcal{D} \models_{\rho[x \,\mapsto\, d]} F$. Since d was chosen arbitrarily, we infer $\mathcal{D} \models_\rho (\forall\, x)\, F$. ■

For completeness, we will show how to embed first-order $\mathcal{CND}$ deductions into $\lambda\mu$–$\mathcal{CND}_1$. Owing to the desugaring of **pick-witness** in terms of **pick-any** and **ex-elim** (which is $\mathsf{qs\text{-}axiom}$ here) given in page 223, we need not be concerned with **pick-witness** deductions. For the remaining constructs, we proceed as follows. First, let $\gamma : \mathcal{V} \to \mathit{Exp}$ be a function from variables to $\lambda\mu$–$\mathcal{CND}_1$ expressions that is the identity almost everywhere (this is possible since the elements of $\mathcal{V}$ are terms, and terms are constants and hence expressions). We extend γ to terms and formulas as follows:

$$\begin{array}{rcl} \widehat{\gamma}(x) & = & \gamma(x) \\ \widehat{\gamma}(a) & = & a \\ \widehat{\gamma}(f(t_1, \ldots, t_n)) & = & \mathbf{app}(\mathsf{f}, \widehat{\gamma}(t_1), \ldots, \widehat{\gamma}(t_n)) \end{array}$$

and

$$\begin{array}{rcl} \widehat{\gamma}(R(t_1, \ldots, t_n)) & = & \mathbf{app}(\mathsf{R}, \widehat{\gamma}(t_1), \ldots, \widehat{\gamma}(t_n)) \\ \widehat{\gamma}(\neg F) & = & \mathbf{app}(\neg, \widehat{\gamma}(F)) \\ \widehat{\gamma}(F_1 \circ F_2) & = & \mathbf{app}(\circ, \widehat{\gamma}(F_1), \widehat{\gamma}(F_2)) \\ \widehat{\gamma}((\forall\, x)\, F) & = & \mathbf{app}(\forall, x, \widehat{\gamma[x \mapsto x]}(F)) \\ \widehat{\gamma}((\exists\, x)\, F) & = & \mathbf{app}(\exists, x, \widehat{\gamma[x \mapsto x]}(F)) \end{array}$$

for $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$. We now define a translation $\mathcal{T}$ from first-order $\mathcal{CND}$ deductions into $\lambda\mu$–$\mathcal{CND}_1$ deductions as follows:

$$\mathcal{T}(D) = T(D, id_{\mathcal{V}}, []))$$

where $id_{\mathcal{V}}$ is the identity on $\mathcal{V}$ and T is defined as follows:

$T(F, \gamma, L) = \widehat{\gamma}(F)$
$T(\mathit{Rule}\ \ F_1, \ldots, F_n, \gamma, L) = \mathbf{dapp}(\mathit{Rule}^*, \widehat{\gamma}(F_1), \ldots, \widehat{\gamma}(F_n))$
$T(\textbf{specialize}\ (\forall\, x)\, F\ \textbf{with}\ t, \gamma, L) = \mathbf{dapp}(\mathsf{uspec}, \widehat{\gamma}((\forall\, x)\, F), \widehat{\gamma}(t))$
$T(\textbf{ex-generalize}\ (\exists\, x)\, F\ \textbf{from}\ t, \gamma, L) = \mathbf{dapp}(\mathsf{egen}, \widehat{\gamma}((\exists\, x)\, F), \widehat{\gamma}(t))$

$T(\textbf{assume } F \textbf{ in } D, \gamma, L) = \textbf{assume } \widehat{\gamma}(F) \textbf{ in } T(D, \gamma, L)$
$T(D_1; D_2, \gamma, L) = T(D_1, \gamma, L); T(D_2, \gamma, L)$
$T(\textbf{pick-any } x \textbf{ in } D, \gamma, L) = \textbf{ dlet } I = \textsf{fresh-var } L \textbf{ in ugen}(I, T(D, \gamma[x \mapsto I], I{::}L))$

where *Rule** is the obvious $\lambda\mu$–$\mathcal{CND}_1$ analogue of the corresponding $\mathcal{CND}$ primitive rule. Also, in the **pick-any** clause, a fresh I should be chosen every time the clause is applied. This is essential in order to ensure that in the translation of a deduction such as

$$\textbf{pick-any } x \textbf{ in } \cdots \textbf{pick-any } y \textbf{ in } \cdots \underline{x} \cdots$$

the identifier that will take the place of the underlined occurence of x refers to the outer eigenvariable. The parameter L, like γ, is only used for desugaring the bodies of **pick-any** deductions: it holds a list of already-allocated eigenvariables (or, more precisely, a list of identifiers whose values will be the said eigenvariables), so that fresh-var can generate distinct entries inside nested universal generalizations. As an example, the translation of

pick-any x **in**
  pick-any y **in**
    assume $x = y$ **in**
      swap $x = y$

is:

dlet $foo_1 = \textsf{fresh-var } []$
in
  generalize-over foo_1 **in**
    dlet $foo_2 = \textsf{fresh-var } [foo_1]$
    in
      generalize-over foo_2 **in**
        assume $foo_1 = foo_2$ **in**
          !swap $foo_1 = foo_2$

We now have:

Lemma 9.29 *If $\beta \vdash_{\mathcal{CND}} D \rightsquigarrow F$ then $\beta \vdash_{\lambda\mu\text{–}\mathcal{CND}_1} \mathcal{T}(D) \rightsquigarrow F$.*

Therefore:

Theorem 9.30 (Completeness) *$\lambda\mu$–$\mathcal{CND}_1$ is complete, i.e., if $\Phi \models F$ then $\Phi \vdash F$.*

9.7.3 The importance of term and formula constructors

Next we define a mapping $[\![\,\cdot\,]\!]$ from terms and formulas to $\lambda\mu$ expressions that will clarify how term and formula constructors may be used to build terms and formulas. We begin with terms:

$$\begin{array}{rcl} [\![x]\!] & = & x \\ [\![c]\!] & = & c \\ [\![f(t_1,\ldots,t_n)]\!] & = & \mathbf{app}(\mathsf{f},[\![t_1]\!],\ldots,[\![t_n]\!]) \end{array}$$

For formulas we set:

$$\begin{array}{rcl} [\![R(t_1,\ldots,t_n)]\!] & = & \mathbf{app}(\mathsf{R},[\![t_1]\!],\ldots,[\![t_n]\!]) \\ [\![\neg F]\!] & = & \mathbf{app}(\neg,[\![F]\!]) \\ [\![F_1 \circ F_2]\!] & = & \mathbf{app}(\circ,[\![F_1]\!],[\![F_2]\!]) \\ [\![(\forall\, x)\, F]\!] & = & \mathbf{app}(\forall, x, [\![F]\!]) \\ [\![(\exists\, x)\, F]\!] & = & \mathbf{app}(\exists, x, [\![F]\!]). \end{array}$$

for $\circ \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$. Now a straightforward induction on t and F will show:

Lemma 9.31 *For all t, F, and β, $\beta \vdash [\![t]\!] \rightsquigarrow t$, $\beta \vdash [\![F]\!] \rightsquigarrow F$.*

To make matters concrete, let us fix a signature $\mathbf{\Omega}_1 = (\mathcal{C},\mathcal{F},\mathcal{R})$ with $\mathcal{C} = \{a,b\}$, $\mathcal{F} = \{f,g,h\}$, and $\mathcal{R} = \{P,Q,R\}$, where the arities of f, g, P and Q are 1, while those of h and R are 2; and let us take $\mathcal{V} = \{x_1,x_2,x_3,\ldots\}$, with the natural well-ordering $x_1 \prec_{\mathcal{V}} x_2 \prec_{\mathcal{V}} x_3 \prec_{\mathcal{V}} \cdots$. Then the reader will verify that, in any β, the expression

$$\wedge\ \ (\mathsf{R}\ \ x_1\ \ a)\ \ (\mathsf{P}\ \ x_2) \tag{9.46}$$

will produce the formula $R(x_1,a) \wedge P(x_2)$. (Recall that by virtue of the conventions we laid down in Section 8.6.1, expression 9.46 is an abbreviation for

$$\mathbf{app}(\wedge, \mathbf{app}(\mathsf{R}, x_1, a), \mathbf{app}(\mathsf{P}, x_2)).$$

To conform with custom, in this section we will also use infix abbreviations for the primitive functions $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$, writing, for example, $(E_1 \wedge E_2)$ instead of the usual s-expression abbreviation $(\wedge\ \ E_1\ \ E_2)$. As usual, outer pairs of parentheses will be omitted, so that 9.46, for instance, would be written in infix as $(\mathsf{R}\ \ x_1\ \ a) \wedge (\mathsf{P}\ \ x_2)$.)

But term and formula constructors are functions and can thus be applied to arbitrarily complicated phrases, not just to constants; in practice this flexibility is very useful. For instance, in any assumption base the expression

$$\mathsf{R}\ \ ((\lambda\, u\,.\,u)\ \ x_5)\ \ (\mathbf{let}\ \mathit{ind} = a\ \mathbf{in}\ \mathit{ind})$$

will result in the atomic $\mathbf{\Omega}_1$-formula

$$R(x_5, a).$$

A more useful example: when writing out formulas with many leading occurences of the same quantifier, it is tiresome to write out each occurence separately. Instead of writing, say,

$$(\forall\, x_1)\,(\forall\, x_2)\,(\forall\, x_3)\,(\forall\, x_4)\, F \tag{9.47}$$

we would like to write something like

$$(\forall^* \,[x_1, x_2, x_3, x_4]\; F) \tag{9.48}$$

where all the quantified variables are assembled together in a single list. In fact we can express $\forall^*$ as a binary $\lambda\mu$–$\mathcal{CND}_1$ function, in such a way that 9.48 will be a function call that will *produce* the formula 9.47! Specifically:

$\forall^* = \lambda\, \textit{var-list}, F\, .$
 match *var-list*
  $[] \Longrightarrow F$
  $v :: \textit{rest-vars} \Longrightarrow \forall\, v\; (\forall^* \;\; \textit{rest-vars} \;\; F)$

Moreover, because the $\lambda\mu$-calculus is higher-order, term and formula constructors can themselves become bound to identifiers and passed as arguments to functions or methods, or be returned as the results of functions. For example, the expression

let $\textit{neg} = \neg$
  $\textit{var} = x_8$
  $Q \;\; = \forall$
in
 $\textit{neg}\;\; (Q\;\; ((\lambda\, u\, .\, u)\;\; x_8)\;\; (\mathsf{R}\;\; \textit{var}\;\; x_8))$

will return the $\mathbf{\Omega}_1$-formula

$$\neg(\forall\, x_8)\, R(x_8, x_8).$$

This also allows for remarkably expressive and succinct patterns. For instance, the pattern $\mathsf{P}\;\; (\mathsf{h}\;\; a\;\; \textit{var})$ will match the atomic formula $P(h(a, x_1))$ with the binding $\textit{var} \mapsto x_1$, while the pattern

$$(\forall\;\; \textit{var}\;\; (\textit{con}\;\; (\mathsf{P}\;\; \textit{var})\;\; (\textit{rel}\;\; t\;\; (\mathsf{h}\;\; a\;\; (\textit{fun}\;\; \textit{var})))))$$

will match the formula

$$(\forall\, x_1)\, [P(x_1) \Rightarrow R(x_8, h(a, g(x_1)))]$$

yielding the bindings

$$var \mapsto x_1, con \mapsto \Rightarrow, rel \mapsto \mathsf{R}, t \mapsto x_8, fun \mapsto \mathsf{g}.$$

This type of pattern matching is found in Athena (see the relevant discussion in page 27).

Before continuing we should emphasize that the variables of $\mathcal{V}$ are *constants* in $\lambda\mu$–$\mathcal{CND}_1$. This point is of critical importance, as most primitive functions (term constructors, ∀, ∃, the function sub, etc.), as well as primitive methods such as uspec and the special deductive form **ugen**, all rely on the ability to pass around and manipulate variables as raw data. This point also marks an essential difference between $\lambda\mu$ systems and Curry-Howard-based frameworks such as LF (where the variables of the object logic are variables in the meta-logic), and has some interesting notational consequences. For example, suppose again that $\mathcal{V} = \{x_1, x_2, \ldots\}$. Then an expression such as $\lambda\, x_2 \,.\, x_2$ is syntactically invalid, since the variable $x_2 \in \mathcal{V}$ is a constant, and a constant cannot be used as the parameter of a λ-abstraction (recall that constants c are supposed to be syntactically distinct from identifiers I). In Athena the distinction is made visually clear by fixing $\mathcal{V}$ to consist of all and only those expressions of the form ?I, where, by fiat, no identifier can begin with ?.

9.7.4 Syntax sugar

The forms **assume** $I = M$ **in** D, **suppose-absurd** M **in** D, and

$$\textbf{suppose-absurd } I = M \textbf{ in } D$$

are introduced as syntax sugar in the manner of $\lambda\mu$–$\mathcal{CND}_0$. Additionally, we define two new syntax forms

$$\textbf{pick-any } I_1, \ldots, I_n \textbf{ in } D \tag{9.49}$$

and **pick-witness** I **for** E **in** D as syntax sugar; they mimic the corresponding constructions of first-order $\mathcal{CND}$. For the former, let $x_1, \ldots, x_k$ be all and only the variables that occur in D. Then 9.49 is taken as an abbreviation for the deduction

$$\begin{array}{l}
\textbf{dlet } I_1 = \ \mathsf{fresh\text{-}var}\ [x_1, \ldots, x_k] \\
\qquad I_2 = \ \mathsf{fresh\text{-}var}\ [x_1, \ldots, x_k, I_1] \\
\qquad\quad \vdots \\
\qquad I_n = \ \mathsf{fresh\text{-}var}\ [x_1, \ldots, x_k, I_1, \ldots, I_{n-1}] \\
\textbf{in} \\
\quad \textbf{ugen}(I_1, \textbf{ugen}(I_2, \ldots, \textbf{ugen}(I_n, D) \cdots))
\end{array}$$

$$
\begin{array}{l}
\textbf{dmatch } E \\
\quad (\exists\ v\ B) \implies \textbf{dlet } F = \textbf{pick-any } I \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{assume } \{v \mapsto I\}\, B \textbf{ in} \\
\qquad\qquad\qquad\qquad\qquad\qquad D \\
\qquad\qquad\qquad\quad \textbf{in} \\
\qquad\qquad\qquad\qquad \textbf{dmatch } F \\
\qquad\qquad\qquad\qquad\quad (\forall\ w\ (F_1 \Rightarrow F_2)) \implies \\
\qquad\qquad\qquad\qquad\qquad \textbf{dlet } G = !\mathsf{qs\text{-}axiom}\ \ F \Rightarrow (\exists\ w\ (F_1 \Rightarrow F_2)) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad !\mathsf{mp}\ (!\mathsf{mp}\ G\ F)\ (\exists\ v\ B)
\end{array}
$$

Figure 9.9: The desugaring of **pick-witness** I **for** E **in** D.

The following lemma shows that this desugaring captures the expected semantics of **pick-any**:

Lemma 9.32 *$\beta \vdash$ **pick-any** $I_1, \ldots, I_n$ **in** $D \rightsquigarrow (\forall x_1)(\forall x_2) \cdots (\forall x_n) F$ whenever*

$$\beta \vdash D[x_1, \ldots, x_n / I_1, \ldots, I_n] \rightsquigarrow F$$

for distinct variables $x_1, \ldots, x_n$ that occur neither in D nor in $FV(\beta)$.

Proof: Immediate by the semantics of **ugen**, fresh-var, and the desugaring of **dlet**. ■

The desugaring of deductions of the form

$$\textbf{pick-witness } I \textbf{ for } E \textbf{ in } D \tag{9.50}$$

is a bit trickier, but based on the same idea as the $\mathcal{CND}$ desugaring of **pick-witness** in terms of **pick-any** and **ex-elim** given in page 223 (with qs-axiom here playing the role of **ex-elim**). It appears in Figure 9.9. We state the following without proof:

Lemma 9.33 *If $\beta \vdash E \rightsquigarrow (\exists x) F$ and*

$$\beta \cup \{\{x \mapsto z\}\, F\} \vdash D[z/I] \rightsquigarrow G$$

whenever z does not occur free in $\beta \cup \{(\exists x) F, G\}$, then

$$\beta \cup \{(\exists x) F\} \vdash \textbf{pick-witness } I \textbf{ for } E \textbf{ in } D \rightsquigarrow G.$$

Let us work out a detailed example, with $\mathbf{\Omega}_1$ and $\mathcal{V} = \{x_1, x_2, \ldots\}$. We will show that

$$\emptyset \vdash D \rightsquigarrow (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)] \Rightarrow (\forall\, x_1)\, P(x_1)$$

where D is the deduction

$$\begin{array}{l}\textbf{assume } F = (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)] \textbf{ in}\\ \quad \textbf{pick-any } i \textbf{ in}\\ \qquad !\mathsf{left\text{-}and}\ \ (!\mathsf{uspec}\ \ F\ \ i).\end{array}$$

Now D desugars into

$$\begin{array}{l}D_1 = \textbf{dlet } F = (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)]\\ \qquad\quad \textbf{in}\\ \qquad\qquad \textbf{assume } F \textbf{ in}\\ \qquad\qquad\quad \textbf{pick-any } i \textbf{ in}\\ \qquad\qquad\qquad !\mathsf{left\text{-}and}\ \ (!\mathsf{uspec}\ \ F\ \ i)\end{array}$$

and desugaring the **dlet** gives

$$\begin{array}{l}D_2 = \textbf{dapp}(\mu\, F\,.\, \textbf{assume } F \textbf{ in}\\ \qquad\qquad\qquad\qquad \textbf{pick-any } i \textbf{ in}\\ \qquad\qquad\qquad\qquad\quad !\mathsf{left\text{-}and}\ \ (!\mathsf{uspec}\ \ F\ \ i),\\ \qquad\qquad (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)]).\end{array}$$

Finally, desugaring the **pick-any** yields

$$\begin{array}{l}D_3 = \textbf{dapp}(\mu\, F\,.\, \textbf{assume } F \textbf{ in}\\ \qquad\qquad\qquad\qquad \textbf{dapp}(\mu\, i\,.\, \textbf{ugen}(i, !\mathsf{left\text{-}and}\ \ (!\mathsf{uspec}\ \ F\ \ i)), (\mathsf{fresh\text{-}var}\ \ [])),\\ \qquad\qquad (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)]).\end{array}$$

Thus our goal becomes to show

$$\emptyset \vdash D_3 \rightsquigarrow (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)] \Rightarrow (\forall\, x_1)\, P(x_1) \tag{9.51}$$

and we accomplish this with the following $\lambda\mu$-derivation. First, letting

$$\begin{array}{l}D_4 = \textbf{assume } (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)] \textbf{ in}\\ \qquad\quad \textbf{dapp}(\mu\, i\,.\, \textbf{ugen}(i, !\mathsf{left\text{-}and}\ \ (!\mathsf{uspec}\ \ (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)]\ \ i)),\\ \qquad\qquad (\mathsf{fresh\text{-}var}\ \ []))\end{array}$$

we have

$$1. \quad \emptyset \vdash D_3 \rightsquigarrow D_4 \quad [\text{R2}].$$

Furthermore, letting $\beta_1 = \{(\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\}$, we have

2. $\beta_1 \vdash (\mathsf{fresh\text{-}var}\ \ []) \rightsquigarrow x_1 \quad \delta\text{-rule}$
3. $\beta_1 \vdash \mathbf{dapp}(\mu\, i\,.\,\mathbf{ugen}(i, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ i)), (\mathsf{fresh\text{-}var}\ \ [])) \rightsquigarrow$ $\mathbf{dapp}(\mu\, i\,.\,\mathbf{ugen}(i, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ i)), x_1) \quad 2, [\text{R6}]$
4. $\beta_1 \vdash \mathbf{dapp}(\mu\, i\,.\,\mathbf{ugen}(i, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ i)), x_1) \rightsquigarrow$ $\mathbf{ugen}(x_1, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ x_1)) \quad [\text{R2}]$
5. $\beta_1 \vdash \mathbf{dapp}(\mu\, i\,.\,\mathbf{ugen}(i, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ i)), (\mathsf{fresh\text{-}var}\ \ [])) \rightsquigarrow$ $\mathbf{ugen}(x_1, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ x_1)) \quad 3, 4, [\text{R11}]$
6. $\beta_1 \vdash \mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ x_1 \rightsquigarrow P(x_1) \wedge Q(x_1) \quad \delta\text{-rule}$
7. $\beta_1 \cup \{P(x_1) \wedge Q(x_1)\} \vdash \mathbf{!}\mathsf{left\text{-}and}\ \ P(x_1) \wedge Q(x_1) \rightsquigarrow P(x_1) \quad \delta\text{-rule}$
8. $\beta_1 \vdash \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ x_1) \rightsquigarrow P(x_1) \quad 6, 7, [\text{R7}]$
9. $\beta_1 \vdash \mathbf{ugen}(x_1, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ x_1)) \rightsquigarrow$ $(\forall\, x_1)\, P(x_1) \quad 8, [\mathbf{ugen}]$
10. $\beta_1 \vdash \mathbf{dapp}(\mu\, i\,.\,\mathbf{ugen}(i, \mathbf{!}\mathsf{left\text{-}and}\ \ (\mathbf{!}\mathsf{uspec}\ \ (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]\ \ i)), (\mathsf{fresh\text{-}var}\ \ [])) \rightsquigarrow$ $(\forall\, x_1)\, P(x_1) \quad 5, 9, [\text{R11}]$
11. $\emptyset \vdash D_4 \rightsquigarrow (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)] \Rightarrow (\forall\, x_1)\, P(x) \quad 10, [\mathbf{assume}]$
12. $\emptyset \vdash D_3 \rightsquigarrow (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)] \Rightarrow (\forall\, x_1)\, P(x) \quad 1{,}11, [\text{R11}]$

which is precisely our goal 9.51.

9.7.5 Examples

We continue with a $\lambda\mu$–$\mathcal{CND}_1$ deduction of the tautology

$$[(\forall\, x_1)\, P(x_1) \wedge (\forall\, x_1)\, Q(x_1)] \Rightarrow (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]$$

where we again take $\mathbf{\Omega}_1$ as our signature and $\mathcal{V} = \{x_1, x_2, \ldots\}$. This tautology was proved in $\mathcal{CND}$ in Section 6.3. Compare that proof with the following one in $\lambda\mu$–$\mathcal{CND}_1$:

$$\boxed{[(\forall\, x_1)\, P(x_1) \wedge (\forall\, x_1)\, Q(x_1)] \Rightarrow (\forall\, x_1)\,[P(x_1) \wedge Q(x_1)]}$$

Proof:

assume $H = [(\forall\, x_1)\, P(x_1) \wedge (\forall\, x_1)\, Q(x_1)]$ **in**
  pick-any i **in**

$$
\begin{array}{l}
\textbf{dlet } S_1 = \texttt{!uspec } (\texttt{!left-and } H)\ i \\
\qquad\ \ S_2 = \texttt{!uspec } (\texttt{!right-and } H)\ i \\
\textbf{in} \\
\ \ \texttt{!both } S_1\ S_2
\end{array}
$$

or with annotated conclusions:

$$
\begin{array}{l}
\textbf{assume } H = [(\forall\, x_1)\, P(x_1) \wedge (\forall\, x_1)\, Q(x_1)] \textbf{ in} \\
\quad (\forall\, x_1)\, [P(x_1) \wedge Q(x_1)] \textbf{ by } \textbf{pick-any } i \textbf{ in} \\
\qquad\qquad \textbf{dlet } S_1 = (\mathsf{P}\ i) \textbf{ by } \texttt{!uspec } (\texttt{!left-and } H)\ i \\
\qquad\qquad\qquad\ \ S_2 = (\mathsf{Q}\ i) \textbf{ by } \texttt{!uspec } (\texttt{!right-and } H)\ i \\
\qquad\qquad \textbf{in} \\
\qquad\qquad\ \ (\mathsf{P}\ i) \wedge (\mathsf{Q}\ i) \textbf{ by } \texttt{!both } S_1\ S_2
\end{array}
$$

The reader will observe that there is nothing special in the foregoing proof about the relation symbols P and Q, or about their arity, or about the specific variable x_1. In other words, there is nothing special about the specific vocabulary: the signature $\mathbf{\Omega}_1$ and the variables $\mathcal{V} = \{x_1, x_2, \ldots\}$. The same reasoning could be applied to derive the tautology

$$[(\forall\, x)\, F_1 \wedge (\forall\, x)\, F_2] \Rightarrow (\forall\, x)\, [F_1 \wedge F_2]$$

where F_1 and F_2 are any formulas, built over an arbitrary signature, and x is an arbitrary variable. Accordingly, we inquire whether it is possible to write a *method*, in strict style (see the discussion in page 336), that will take any given premise of the form $(\forall\, x)\, F_1 \wedge (\forall\, x)\, F_2$, irrespective of the specific identity of F_1, F_2, and x, and will reason as above in order to deduce the conclusion $(\forall\, x)\, [F_1 \wedge F_2]$. The following method accomplishes this:

$$
\begin{array}{l}
m_1 = \mu\, \textit{premise}\,. \\
\ \textbf{dmatch } \textit{premise} \\
\quad (\forall\, v\ F_1) \wedge (\forall\, v\ F_2) \implies \textbf{pick-any } i \textbf{ in} \\
\qquad\qquad \textbf{dlet } S_1 = \texttt{!uspec } (\texttt{!left-and } \textit{premise})\ i \\
\qquad\qquad\qquad\ \ S_2 = \texttt{!uspec } (\texttt{!right-and } \textit{premise})\ i \\
\qquad\qquad \textbf{in} \\
\qquad\qquad\ \ \texttt{!both } S_1\ S_2
\end{array}
$$

This will work for any signature because v, F_1, and F_2 are $\lambda\mu$ pattern variables that will get appropriately instantiated depending on the specific input; the only constants that are used are the primitive functions $\wedge$ and $\forall$. Schematically, we may depict the "interface" of m_1 as follows:

$$\frac{(\forall\, x)\, F_1 \wedge (\forall\, x)\, F_2}{(\forall\, x)\, [F_1 \wedge F_2]} \quad [m_1]$$

Continuing in the same vein, we formulate methods that capture several useful "derived" inference rules (most of these were proved hypothetically, as tautologies, in $\mathcal{CND}$, Section 6.3, pages 157–163):

$$\frac{(\exists\, x)\, [F_1 \wedge F_2]}{(\exists\, x)\, F_1 \wedge (\exists\, x)\, F_2} \quad [m_2]$$

$m_2 = \mu\, premise\,.$
dmatch *premise*
  $\exists\, v\ (F_1 \wedge F_2) \implies$ **pick-witness** w **for** *premise* **in**
    dlet $F\ = \{v \mapsto w\}\,(F_1 \wedge F_2)$
      $\_\ = \{v \mapsto w\}\, F_1$ **by** !left-and F
      $S_1 = (\exists\, v\ F_1)$ **by** !egen $(\exists\, v\ F_1)\ w$
      $\_\ = \{v \mapsto w\}\, F_2$ **by** !right-and F
      $S_2 = (\exists\, v\ F_2)$ **by** !egen $(\exists\, v\ F_2)\ w$
    in
     $(\exists\, v\ F_1) \wedge (\exists\, v\ F_2)$ **by** !both $S_1\ S_2$

$$\frac{(\forall\, x)\, \neg\neg F}{(\forall\, x)\, F} \quad [m_3]$$

$m_3 = \mu\, premise\,.$
dmatch *premise*
  $(\forall\, v\ \neg\neg F) \implies$ **pick-any** i **in**
    !dn (!uspec *premise* i)

$$\frac{\neg(\exists\, x)\, F}{(\forall\, x)\, \neg F} \quad [m_4]$$

$m_4 = \mu\, premise\,.$
dmatch *premise*
  $\neg\,(\exists\, v\ F) \implies$ **pick-any** i **in**
    suppose-absurd $\{v \mapsto i\}\, F$ **in**

$$\text{!absurd}\ \ (\text{!egen}\ \ (\exists\, v\ F)\ \ i)\ \ \textit{premise}$$

$$\frac{(\forall\, x)\, \neg F}{\neg(\exists\, x)\, F}\quad [m_5]$$

$m_5 = \mu\, \textit{premise}\,.$
dmatch *premise*
$(\forall\, v\ \neg F) \implies$ **suppose-absurd** $H = (\exists\, v\ F)$ **in**
pick-witness w **for** H **in**
$\text{!absurd}\ \ \{v \mapsto w\}\, F\ \ (\text{!uspec}\ \ \textit{premise}\ \ w)$

$$\frac{\neg(\forall\, x)\, F}{(\exists\, x)\, \neg F}\quad [m_6]$$

$m_6 = \mu\, \textit{premise}\,.$
dmatch *premise*
$\neg\,(\forall\, v\ F) \implies$ **dlet** $S =$ **suppose-absurd** $H = \neg\,(\exists\, v\ \neg\, F)$ **in**
$\text{!absurd}\ \ (\text{!}m_3\ \ (\text{!}m_4\ \ H))\ \ \textit{premise}$
in
$\text{!dn}\ \ S$

Notice how the construction of new methods is facilitated by the use of previously defined methods; this drastically reduces the size of new methods. This is precisely the sort of "abstraction" and "task decomposition" that we were talking about in the introduction, and in Section 2.3.

$$\frac{(\exists\, x)\, \neg F}{\neg(\forall\, x)\, F}\quad [m_7]$$

$m_7 = \mu\, \textit{premise}\,.$
dmatch *premise*
$(\exists\, v\ \neg\, F) \implies$ **suppose-absurd** $H = (\forall\, v\ F)$ **in**
pick-witness w **for** *premise* **in**
$\text{!absurd}\ \ (\text{!uspec}\ \ H\ \ w)\ \ (\neg\,\{v \mapsto w\}\, F)$

$$\frac{(\forall\, x)\, F}{(\exists\, x)\, F}\ [m_8]$$

$m_8 = \mu\, premise\, .$
dmatch *premise*
$(\forall\, v\ F) \implies$ **dlet** $S_1 =$ **suppose-absurd** $H = \neg\,(\exists\, v\ F)$ **in**
dlet $S_2 = (\forall\, v\ \neg\, F)$ **by** $!m_4\ \ H$
in
!absurd (!uspec *premise* v) (!uspec $S_2\ \ v$)
in
!dn S_1

$$\frac{(\forall\, x)\, [F \Rightarrow G]}{(\forall\, x)\, F \Rightarrow (\forall\, x)\, G}\ [m_9]$$

$m_9 = \mu\, premise\, .$
dmatch *premise*
$(\forall\, v\ (F \Rightarrow G)) \implies$ **assume** $H = (\forall\, v\ F)$ **in**
pick-any i **in**
!mp (!uspec *premise* i) (!uspec $H\ \ i$)

$$\frac{(\forall\, x)\, [F \Rightarrow G]}{(\forall\, x)\, F \Rightarrow (\exists\, x)\, G}\ [m_{10}]$$

$m_{10} = \mu\, premise\, .$
dmatch *premise*
$(\forall\, v\ (F \Rightarrow G)) \implies$ **assume** $H = (\forall\, v\ F)$ **in**
$!m_8$ (!mp ($!m_9$ *premise*) H)

Note that m_{10} abstracts over the $\mathcal{CND}$ deduction of

$$(\forall\, x)\, [P(x) \Rightarrow Q(x)] \Rightarrow [(\forall\, x)\, P(x) \Rightarrow (\forall\, x)\, Q(x)]$$

shown in page 162. The reader should observe how much smaller the present method is, and more importantly, how the need for the informal directive "Insert the deduction of $(\forall\, x)\, \neg Q(x)$ form $\neg(\exists\, x)\, Q(x)$" is eliminated here thanks to method calls.

A more detailed, step-by-step version of m_{10}, with annotated conclusions, attests to the expressive versatility of $\lambda\mu$–$\mathcal{CND}_1$:

$m_{10} = \mu\, premise\,.$
dmatch *premise*
$(\forall\, v\ (F \Rightarrow G))\ \Longrightarrow$ **assume** $H = (\forall\, v\ F)$ **in**
dlet $S_1 = (\forall\, v\ F) \Rightarrow (\forall\, v\ G)$ **by** $!m_9\ \ premise$
$S_2 = (\forall\, v\ G)$ **by** !mp $S_1\ \ H$
in
$(\exists\, v\ G)$ **by** $!m_8\ \ S_2$

We conclude with a recursive method *uspec** that takes a premise

$$(\forall\, x_1)\,(\forall\, x_2)\,\cdots(\forall\, x_n)\,F$$

with an arbitrarily long chain of n leading universal quantifiers, and a list of n terms $[t_1, \ldots, t_n]$, and derives the formula obtained by successively specializing x_1 with t_1, x_2 with t_2, …, x_n with t_n:

*uspec** $= \lambda\, F, term\text{-}list\,.$
dmatch *term-list*
$[]\ \Longrightarrow$!claim F
$t :: rest\text{-}terms \Longrightarrow$!*uspec** (!*uspec* F t) *more-terms*

Our last example will be of a more conventional mathematical flavor: we will prove that if a binary relation is irreflexive and transitive then it is also assymetric. The formal definitions of these concepts are as follows:

$$R \text{ is irreflexive:} \quad (\forall\, x)\,\neg R(x,x) \tag{9.52}$$

$$R \text{ is assymetric:} \quad (\forall\, x)\,(\forall\, y)\,[R(x,y) \Rightarrow \neg R(y,x)] \tag{9.53}$$

$$R \text{ is transitive:} \quad (\forall\, x)\,(\forall\, y)\,(\forall\, z)\,[R(x,y) \wedge R(y,z) \Rightarrow R(x,z)] \tag{9.54}$$

First let us see how we might go about this task informally.

Taking as working assumptions that R is irreflexive and transitive, we need to show that for any a and b, if $R(a,b)$ then $\neg R(b,a)$. Therefore, decomposing our task in a top-down manner, we see that our proof must have the following structure:

pick any a, b
assume $R(a,b)$
D

where D is a deduction that must establish $\neg R(b,a)$, and may freely use the assumptions that R is irreflexive and transitive, in addition to the hypothesis $R(a,b)$. Our task now has been reduced to discovering such a D.

Since the conclusion of D must be a negation (to wit, $\neg R(b,a)$), and negations are often introduced via reasoning by contradiction (**suppose-absurd**), we proceed by taking $R(b,a)$ as a hypothesis in the hope of discovering an inconsistency:

pick any a, b
  assume $R(a, b)$
    suppose, by way of contradiction, that $R(b, a)$
      (i) From $R(b, a)$, $R(a, b)$, and the fact that R is transitive, we get $R(b, b)$.
      (ii) But R is irreflexive, hence we must have $\neg R(b, b)$.
      (iii) $R(b, b)$ and $\neg R(b, b)$ constitute a contradiction—**false**!

Figure 9.10: An informal proof showing that if R is irreflexive and transitive, then it is assymetric.

pick any a, b
  assume $R(a, b)$
    suppose, by way of contradiction, that $R(b, a)$
      D'

and we are now left with the task of discovering a D' that manages to derive the absurdity **false** on the basis of the current hypotheses. This is not difficult at all: by the assumption of transitivity, the hypotheses $R(b, a)$ and $R(a, b)$ give $R(b, b)$—but this contradicts the assumption that R is irreflexive. Thus the full proof takes the form shown in Figure 9.10.

Naturally, this informal proof skips some "obvious" steps, involving mainly the instantiation of the universally quantified statements 9.52—9.54 with the eigenvariables a and b, and a bit of propositional reasoning. Consider inference (i), for example, which purports to derive $R(b, b)$ from

(A) $R(b, a)$;

(B) $R(a, b)$; and

(C) "the fact that R is transitive".

Formally, the phrase "the fact that R is transitive" boils down to no more and no less than "the formula 9.54 is in the assumption base". To actually obtain the conclusion $R(b, b)$ from this along with (A) and (B), we need to instantiate the formula 9.54 three successive times with $x \mapsto b, y \mapsto a$, and $z \mapsto b$ to obtain

$$R(b, a) \wedge R(a, b) \Rightarrow R(b, b) \tag{9.55}$$

then use $\wedge$-introduction on (A) and (B) to get

$$R(b, a) \wedge R(a, b) \tag{9.56}$$

and finally use Modus Ponens on 9.55 and 9.56. Similar manipulation must occur in step (ii) with formula 9.52 and the eigenvariable b. Now in a formal proof these steps must appear explicitly at some point, but they can be teased apart in separate and clearly delineated parts so as not to obscure the main idea of the argument. We will see that the abstraction mechanism of methods is ideal for such decomposition.

In order to arrive at the right abstractions, we analyze the informal proof in a little more depth. Upon closer inspection, we see that what is really happening at (i) is this: we are hiding all of the aforementioned details of instantiating 9.54 with the appopriate eigenvariables, using $\wedge$-introduction and Modus Ponens, etc., by tacitly invoking a "derived" inference rule of the form:

$$\frac{R(t_1,t_2) \qquad R(t_2,t_3)}{R(t_1,t_3)}\ [\mathit{transitivity}]$$
$$\text{whenever } R \text{ is transitive}$$

that is presumed to "fill in" all those tedious details. Likewise, in (ii), we are tacitly using a nullary rule of the form

$$\frac{}{\neg R(t,t)}\ [\mathit{irreflexivity}]$$
$$\text{whenever } R \text{ is irreflexive.}$$

All we do in (ii) is apply this rule with b in place of t. The responsibility of actually performing a specialization of the universal quantification 9.52 is relegated to the internal workings of *irreflexivity*; all we need to do is invoke the rule.

Let us see now how we can transcribe these ideas into $\lambda\mu$–$\mathcal{CND}_1$ code. First we observe that because atomic formula constructors can be passed around and manipulated as data, we can abstract our main proof over R; we can then apply it to a specific binary relation by passing it some particular constructor R. To illustrate this kind of manipulation, consider the following function:

$\mathit{irreflexive} = \lambda\, R\,.$
$\ \forall\, x \neg\, (R\ \ x\ \ x)$

This is a function that takes any given binary atomic formula constructor and produces a formula (of course, since this is a function and not a method, the produced formula does not have to be a logical consequence of the assumption base). For instance, suppose that our signature contained a binary relation symbol L, so that we had a binary primitive function L with the semantics

$$\beta \vdash \mathbf{app}(\mathsf{L}, t_1, t_2) \rightsquigarrow L(t_1, t_2).$$

Then the function call **app**(*irreflexive*, L) would produce the formula $(\forall\, x)\,\neg L(x, x)$. Likewise, the functions

$$\begin{array}{l} \mathit{assymetric} = \lambda\, R\,. \\ \;\forall\, x_1\ (\forall\, x_2\ ((R\ \ x_1\ \ x_2) \Rightarrow \neg\,(R\ \ x_2\ \ x_1))) \end{array}$$

and

$$\begin{array}{l} \mathit{transitive} = \lambda\, R\,. \\ \;\forall\, x_1\ (\forall\, x_2\ (\forall\, x_3\ (((R\ \ x_1\ \ x_2) \wedge (R\ \ x_2\ \ x_3)) \Rightarrow (R\ \ x_1\ \ x_3)))) \end{array}$$

could be applied to, say L, to produce, respectively, the formulas

$$(\forall\, x_1)\,(\forall\, x_2)\,[L(x_1, x_2) \Rightarrow \neg L(x_2, x_1)]$$

and

$$(\forall\, x_1)\,(\forall\, x_2)\,(\forall\, x_3)\,[L(x_1, x_2) \wedge L(x_2, x_3) \Rightarrow L(x_1, x_3)].$$

The task before us, then, is to write a unary method *prove-assymetric*, that has one parameter R, and which derives formula 9.53 whenever formulas 9.52 and 9.54 hold (are in the assumption base). Note that we are choosing to write the method in "strict style", counting on 9.52 and 9.54 to be in the assumption base at the time of invocation. Of course we could also write it in hypothetical style, by explicitly assuming 9.52 and 9.54; in that case the conclusion would be a more long-winded conditional to the effect that "if 9.52, then if 9.54, then 9.53"; see page 336 for the pros and cons of each alternative.

Now, following our analysis above, suppose we have a binary method *transitivity* that takes two premises of the form $R(t_1, t_2)$ and $R(t_2, t_3)$ and, on the assumption that 9.54 holds, derives $R(t_1, t_3)$; and a binary method *irreflexivity* that takes an atomic binary constructor R and a term t and, on the assumption that 9.52 holds, derives the formula $\neg R(t, t)$. Then we can express *prove-assymetric* as follows:

$$\begin{array}{l}
\mathit{prove\text{-}assymetric} = \mu\, R\,. \\
\;\textbf{pick-any}\ a, b\ \textbf{in} \\
\quad\textbf{assume}\ (R\ \ a\ \ b)\ \textbf{in} \\
\qquad\textbf{suppose-absurd}\ (R\ \ b\ \ a)\ \textbf{in} \\
\qquad\quad\textbf{dlet}\ S = (R\ \ b\ \ b)\ \textbf{by}\ !\mathit{transitivity}\ \ (R\ \ b\ \ a)\ \ (R\ \ a\ \ b) \\
\qquad\qquad\quad S' = \neg\,(R\ \ b\ \ b)\ \textbf{by}\ !\mathit{irreflexivity}\ \ R\ \ b \\
\qquad\quad\textbf{in} \\
\qquad\qquad !\mathsf{absurd}\ \ S\ \ S'
\end{array}$$

The reader should compare this with the informal proof in Figure 9.10.

The methods *transitivity* and *irreflexivity* that perform the tedious quantifier manipulations and the propositional reasoning we discussed earlier are written as follows:

$transitivity = \mu\, premise_1, premise_2\, .$
;; This method takes two premises of the form $R(t_1, t_2)$ and $R(t_2, t_3)$,
;; where $(\forall\, x)\,(\forall\, y)\,(\forall\, z)\,[R(x,y) \wedge R(y,z) \Rightarrow R(x,z)]$ is assumed to hold, and
;; derives the conclusion $R(t_1, t_3)$.
dmatch $[premise_1, premise_2]$
  $[(R\ t_1\ t_2), (R\ t_2\ t_3)] \Longrightarrow$
    dlet $S_1 = \forall^* [x_1, x_2, x_3]\, ((R\ x_1\ x_2) \wedge (R\ x_2\ x_3) \Rightarrow (R\ x_1\ x_3))$
      by !claim $(transitive\ R)$
     $S_2 = ((R\ t_1\ t_2) \wedge (R\ t_2\ t_3)) \Rightarrow (R\ t_1\ t_3)$ **by** !uspec* $S_1\ [t_1, t_2, t_3]$
    in
     $(R\ t_1\ t_3)$ **by** !mp S_2 (!both $premise_1\ premise_2$)

$irreflexivity = \mu\, R, t\, .$
;; This method takes (1) a binary atomic formula constructor R
;; such that $(\forall\, x)\, \neg R(x,x)$ is assumed to hold, and (2) a term t,
;; and derives $\neg R(t,t)$.
dlet $S = \forall\, x_1\, \neg\, (R\ x_1\ x_1)$ **by** !claim $(irreflexive\ R)$
in
 !uspec $S\ t$

For instance, with L as described earlier and

$$\beta = \{(\forall\, x_1)\, \neg L(x_1, x_1), (\forall\, x_1)\,(\forall\, x_2)\,(\forall\, x_3)\,[L(x_1,x_2) \wedge L(x_2,x_3) \Rightarrow L(x_1,x_3)]\}$$

the reader will verify that

$$\beta \vdash !\mathit{prove\text{-}assymetric}\ \mathsf{L}\ \rightsquigarrow (\forall\, x_1)\,(\forall\, x_2)\,[L(x_1,x_2) \Rightarrow \neg L(x_2,x_1)].$$

9.7.6 $\lambda\mu$–$\mathcal{CND}_1$ as a foundation for first-order theories

$\lambda\mu$–$\mathcal{CND}_1$ allows for first-order natural deduction, and as such it is a fundamental system. Many well-known theories can be cast in this framework, simply by fixing a signature and specifying either an appropriate set of axioms β_0 or a collection of primitive methods. By "postulating a set of axioms β_0" we simply mean that we intend to evaluate deductions only in supersets of β_0; that is, we restrict attention to assumption bases that contain the sentences of β_0. Put differently, we take the members of β_0 as "given".

By and large, the two approaches are equivalent. For instance, in Peano arithmetic we could either take the formula

$$(\forall\, n)\,(\forall\, m)\; s(n) = s(m) \Rightarrow n = m \tag{9.57}$$

as an axiom, or, alternatively, cast it as a primitive method that takes two arbitrary terms t_1 and t_2 such that $s(t_1) = s(t_2)$ holds and produces $t_1 = t_2$. This is the strict version of the method. A hypothetical version is also possible that would take t_1 and t_2 and would output the conditional $s(t_1) = s(t_2) \Rightarrow t_1 = t_2$. The two versions are equivalent, in the sense that either method is expressible in terms of the other, owing to **assume** and mp. And both methods are equivalent to taking 9.57 as an axiom, since if we assume that every assumption base contains 9.57 then it is trivial to write either of the two methods, thanks to uspec and mp. And conversely, if we have either method then we can easily write a **pick-any** deduction that will derive 9.57.

However, primitive methods are more powerful than axioms on account of their "input/output" capability. Take induction in Peano arithmetic, for instance. This could be easily captured by a primitive method that took two premises of the form $\{n \mapsto 0\}\, F$ and $(\forall\, n)\, (F \Rightarrow \{n \mapsto s(n)\}\, F)$ and generated the formula $(\forall\, n)\, F$. Without methods, we would have to cast it as an axiom schema, which would essentially translate into an infinite (but recursive) assumption base.

We emphasize, however, that the underlying machinery of $\lambda\mu$–$\mathcal{CND}_1$ would remain the same from theory to theory. In particular, all of the primitive methods of $\lambda\mu$–$\mathcal{CND}_1$ (dn, mp, both, left-either, uspec, egen, etc.), both of its special deductive forms (**assume** and **ugen**), and, a fortiori, all of the forms we introduced as syntax sugar (**pick-any**, **pick-witness**, **suppose-absurd**, etc.), as well of course as the core syntax and semantics of the $\lambda\mu$-calculus (method abstraction and application, etc.) would carry over unchanged. The only new thing would be the signature and a collection of axioms and/or primitive methods.

For instance, if we fix the signature $\mathbf{\Omega} = (\mathcal{C}, \mathcal{F}, \mathcal{R})$ to be $\mathcal{C} = \{0\}$, $\mathcal{F} = \{s, +, -\}$, and $\mathcal{R} = \{\approx\}$, with unary s and binary $+$, $-$, and $\approx$, and we take any set of variables $\mathcal{V}$ we please, then we have the language of $\mathcal{TL}$, Chapter 2. If we also introduce ten primitive methods ref, sym, tran, id, +cong, −cong, com, inv, +assoc, + − assoc with semantics

$$\emptyset \vdash \mathbf{dapp}(\mathsf{ref}, t) \rightsquigarrow t \approx t$$
$$\{t_1 \approx t_2\} \vdash \mathbf{dapp}(\mathsf{sym}, t_1 \approx t_2) \rightsquigarrow t_2 \approx t_1$$
$$\{t_1 \approx t_2, t_2 \approx t_3\} \vdash \mathbf{dapp}(\mathsf{tran}, t_1 \approx t_2, t_2 \approx t_3) \rightsquigarrow t_1 \approx t_3$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{id}, t) \rightsquigarrow t + 0 \approx t$$
$$\{t_1 \approx t_2, t_1' \approx t_2'\} \vdash \mathbf{dapp}(+\mathsf{cong}, t_1 \approx t_2, t_1' \approx t_2') \rightsquigarrow t_1 + t_1' \approx t_2 + t_2$$
$$\{t_1 \approx t_2, t_1' \approx t_2'\} \vdash \mathbf{dapp}(-\mathsf{cong}, t_1 \approx t_2, t_1' \approx t_2') \rightsquigarrow t_1 - t_1' \approx t_2 - t_2$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{com}, t_1, t_2) \rightsquigarrow t_1 + t_2 \approx t_2 + t_1$$
$$\emptyset \vdash \mathbf{dapp}(\mathsf{inv}, t) \rightsquigarrow t - t \approx 0$$
$$\emptyset \vdash \mathbf{dapp}(+\mathsf{assoc}, t_1, t_2, t_3) \rightsquigarrow (t_1 + t_2) + t_3 \approx t_1 + (t_2 + t_3)$$

$$\emptyset \vdash \mathbf{dapp}(+\mathsf{assoc}, t_1, t_2, t_3) \rightsquigarrow (t_1 + t_2) - t_3 \approx t_1 + (t_2 - t_3)$$

(where here we speak of "terms" t rather than "expressions" e) then we have the complete logic $\mathcal{TL}$. The mechanisms for method abstraction, reasoning with universal quantifiers, etc., are all inherited from $\lambda\mu$–$\mathcal{CND}_1$. The reader should now be able to make perfect sense of the details of our discussion in Section 2.3, and in particular of the diplays in Figure 2.7, Figure 2.8, and Figure 2.9.

A
Herbrand terms

In this chapter we present some basic material on Herbrand terms. The terminology and notation that we introduce here are used in our discussion of first-order systems (Chapter 6, Section 9.7, and Section 9.5), as well as in our treatment of unification in Section 9.6.

A.1 Basic concepts

By a *term signature* we will mean a pair $\Sigma = (\mathcal{C}, \mathcal{F})$ consisting of a set $\mathcal{C}$ of *constant symbols* and a disjoint set $\mathcal{F}$ of *function symbols*, where each $f \in \mathcal{F}$ has a unique positive integer n associated with it and known as its *arity*. We write Σ_0 for $\mathcal{C}$ and Σ_n for the set of all function symbols in $\mathcal{F}$ that have arity n. We also assume the existence of a set of *variables* $\mathcal{V}$, disjoint from $\mathcal{C}$ and $\mathcal{F}$. Although no such restriction is necessary in general, in this document we require $\mathcal{V}$ to be countable. We will use the letters x, y, and z as typical variables; the letters a, b, and c as constant symbols; and the letters f, g, and h as function symbols. A pair $(\Sigma, \mathcal{V})$ consisting of a signature Σ and a set of variables $\mathcal{V}$ will be called a *term vocabulary*.

Fix a term vocabulary $(\Sigma, \mathcal{V})$. The set of *Herbrand terms over* Σ *and* $\mathcal{V}$ (or simply "terms over Σ and $\mathcal{V}$") is defined as follows:

- every variable $x \in \mathcal{V}$ is a term over Σ and $\mathcal{V}$;
- every constant symbol $c \in \Sigma_0$ is a term over Σ and $\mathcal{V}$; and

- if $f \in \Sigma_n$ and $t_1, \ldots, t_n$ are terms over Σ and $\mathcal{V}$, $n > 0$, then $f(t_1, \ldots, t_n)$ is a term over Σ and $\mathcal{V}$.

Terms of the form $f(t_1, \ldots, t_n)$ are called *applications*. We say that f is the *top* (or *root*) function symbol of such a term, and that $t_1, \ldots, t_n$ are its *children*. We write $\mathbf{Terms}(\Sigma, \mathcal{V})$ for the set of all terms over Σ and $\mathcal{V}$. The letters s and t will range over terms.

As an example, consider $\Sigma = (\{0\}, \{s, \mathit{plus}, \mathit{times}\})$, where the arity of s is one, the arity of *plus* is two, and the arity of *times* is also two. Let $\mathcal{V} = \{x, y, z\}$. Then $0, \mathit{plus}(x, s(0))$ and $\mathit{times}(\mathit{plus}(s(0), 0), s(s(0)))$ are all terms over Σ and $\mathcal{V}$.

Subterms are defined recursively: every term is a subterm of itself; and t is a subterm of $f(t_1, \ldots, t_n)$ whenever it is a subterm of some t_i.

We frequently use the principle of *structural induction* to prove that a certain claim holds for every $t \in \mathbf{Terms}(\Sigma, \mathcal{V})$. This generally involves three steps: proving that the claim holds for every variable; proving that it holds for every constant symbol; and proving that it holds for every application $f(t_1, \ldots, t_n)$ on the assumption that it holds for $t_1, \ldots, t_n$ (this assumption is the "inductive hypothesis"). Structural induction is merely a stylistic variation of mathematical induction on the natural numbers, as every argument of the above form could also be cast as a conventional inductive argument on the height of a term. Nevertheless, structural induction is more convenient and is usually preferred over its standard counterpart.

Moreover, because terms are freely generated by the above inductive definition (indeed, $\mathbf{Terms}(\Sigma, \mathcal{V})$ is the paradigmatic free algebra) we are allowed to give recursive definitions of functions on terms by pattern-matching against the three clauses of the definition. As a first example, let $Var(t)$ be the set of variables that occur in a term t; e.g., $Var(\mathit{plus}(0, y)) = \{y\}$. We can recursively define this as a function Var from $\mathbf{Terms}(\Sigma, \mathcal{V})$ to the power-set of $\mathcal{V}$ with the following equations:

$$\begin{aligned} Var(x) &= \{x\} \\ Var(c) &= \emptyset \\ Var(f(t_1, \ldots, t_n)) &= \bigcup_{i=1}^{n} Var(t_i) \end{aligned}$$

As another example, we define the *size* of a term t, denoted $SZ(t)$, as follows:

$$\begin{aligned} SZ(x) &= 1 \\ SZ(c) &= 1 \\ SZ(f(t_1, \ldots, t_n)) &= 1 + \sum_{i=1}^{n} SZ(t_i) \end{aligned}$$

The *height* of a term is yet another example of a quantity that can be defined in a similar recursive manner: the height of a variable or constant symbol is zero, while that of an application is one more than the height of the tallest child. Formally, the legitimacy of such function definitions is ensured by the well-known theorem asserting that if A is an algebra that is freely generated by a set X, then any mapping from X to a similar algebra B can be uniquely extended to a homomorphism from A to B [69].

If a term t contains no variables (i.e., $Var(t) = \emptyset$) we say that t is a *ground* or *closed* term. Non-ground terms are called *open*. We write $\mathbf{Terms}(\Sigma)$ for the set of all ground terms over Σ. Accordingly, $\mathbf{Terms}(\Sigma) = \mathbf{Terms}(\Sigma, \emptyset)$.

We define $TDom(t)$, the *domain* of a term t, as a set of lists of positive integers:

$$\begin{array}{rcl} TDom(x) & = & \{[]\} \\ TDom(c) & = & \{[]\} \\ TDom(f(t_1,\ldots,t_n)) & = & \{[]\} \cup \{i{::}p \mid p \in TDom(t_i), i = 1,\ldots,n\} \end{array}$$

Every list $p \in TDom(t)$ represents a *position* (or *address*): a sequence of positive integers such as $[2,1,3]$, indicating the path that one must follow in order to get from the root of the term to the root of a certain subterm [21].[1] These positions can be very helpful in identifying different parts of a term. Specifically, we define a function *TLabel* that takes a term t and a position $p \in TDom(t)$ and produces whatever “label” (symbol or variable) appears in that position. The precise definition is:

$$\begin{array}{rcl} TLabel(x,[]) & = & x \\ TLabel(c,[]) & = & c \\ TLabel(f(t_1,\ldots,t_n),[]) & = & f \\ TLabel(f(t_1,\ldots,t_n), i{::}p) & = & TLabel(t_i,p) \;\; (\text{for } i \in \{1,\ldots,n\}). \end{array}$$

Owing to the free generation of terms and lists, a straightforward induction on t will show that $TLabel(t,p)$ is unique and well-defined for all $p \in TDom(t)$. The value of $TLabel(t,p)$ for $p \notin TDom(t)$ is left unspecified. (An implementation might report an error in such cases to indicate that an invalid position was given.)

[1] Observe that $TDom(t)$ is *upwards-closed*, meaning that if $p_1 \oplus p_2 \in TDom(t)$ then $p_1 \in TDom(t)$; and *left-closed*, meaning that if $p \oplus [n] \in TDom(t)$ then $p \oplus [i] \in TDom(t)$ for all $i < n$. We could alternatively define Herbrand terms as arity-respecting functions from upwards- and left-closed sets of lists of positive integers to $\mathcal{C} \cup \mathcal{F} \cup \mathcal{V}$, which is the course taken by Gallier [27], Courcelle [21], and others, and which would essentially identify a term t with the function $\lambda p \,.\, TLabel(t,p)$. Such definitions are more abstract than ours, since they avoid concrete-representation details such as parentheses and commas, but both approaches give rise to isomorphic term algebras and thus the differences are inessential.

Finally, for any $p \in TDom(t)$, we define *the subterm of t at position p*, denoted t/p, as:

$$\begin{array}{rcl} t/[] & = & t \\ f(t_1, \ldots, t_n)/i{::}q & = & t_i/q. \end{array}$$

A.2 Substitutions

Let $(\Sigma, \mathcal{V})$ be a term vocabulary. A $(\Sigma, \mathcal{V})$-*substitution* (or simply "substitution" when the reference to the vocabulary is not necessary) is a function $\theta : \mathcal{V} \to \mathbf{Terms}(\Sigma, \mathcal{V})$ that is the identity on all but finitely many elements of $\mathcal{V}$; that is, $\theta(x) \neq x$ only for finitely many variables $x \in \mathcal{V}$. The finite set comprised by these variables is called the *support* of θ, and will be denoted by $Supp(\theta)$; that is, $Supp(\theta) = \{x \in \mathcal{V} \mid \theta(x) \neq x\}$. In addition, we define

$$RanVar(\theta) = \bigcup_{x \in Supp(\theta)} Var(\theta(x)).$$

The letters θ, σ, and τ will be used to denote substitutions.

Since a substitution θ is completely determined by its restriction to its support, it is customary to identify θ with the finite set $\{\langle x_1, \theta(x_1)\rangle, \ldots, \langle x_k, \theta(x_k)\rangle\}$, where $\{x_1, \ldots, x_k\} = Supp(\theta)$. We will use the more suggestive notation

$$\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$$

to represent the substitution that maps each x_i to t_i and every other variable to itself. We write $\{\}$ for the substition that has an empty support, that is, the identity function on $\mathcal{V}$. We call this the *empty substitution*.

Any substitution $\theta : \mathcal{V} \to \mathbf{Terms}(\Sigma, \mathcal{V})$ can be uniquely extended to a function

$$\overline{\theta} : \mathbf{Terms}(\Sigma, \mathcal{V}) \to \mathbf{Terms}(\Sigma, \mathcal{V})$$

as follows:

$$\begin{array}{rcl} \overline{\theta}(x) & = & \theta(x) \\ \overline{\theta}(c) & = & c \\ \overline{\theta}(f(t_1, \ldots, t_n)) & = & f(\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n)). \end{array}$$

Computing $\overline{\theta}(t)$ is known as *applying the substitution* θ to t. For instance, if θ is the substitution $\{x \mapsto s(0), y \mapsto z\}$ and t is the term $f(z, x, y)$ then applying θ to t yields the term $f(z, s(0), z)$. In symbols, $\overline{\theta}(f(z, x, y)) = f(z, s(0), z)$. The reader familiar with universal algebra will recognize $\overline{\theta}$ as the unique homomorphic—and in this case, endomorphic—extension of the mapping $\theta : \mathcal{V} \to \mathbf{Terms}(\Sigma, \mathcal{V})$ to the term algebra $\mathbf{Terms}(\Sigma, \mathcal{V})$.

For notational convenience, we sometimes write $\theta\, t$ as an abbreviation for $\overline{\theta}(t)$. E.g., the expression $\{x \mapsto s(0)\}\, f(x, x)$ will stand for $f(s(0), s(0))$, the term obtained by applying $\{x \mapsto s(0)\}$ to $f(x, x)$. Note that some authors prefer to use postfix notation for substitution application, writing $t\,\theta$ where we write $\theta\, t$.

A.2.1 Composing substitutions

Substitutions are functions from $\mathcal{V}$ to $\mathbf{Terms}(\Sigma, \mathcal{V})$, and thus the composition $\sigma \cdot \tau$ is not well-defined. However, $\overline{\sigma} \cdot \tau$ *is* well-defined, and this is in fact how we define the "composition" of σ and τ. Formally, we define a binary operation $\circ$ on the set of all $(\Sigma, \mathcal{V})$-substitutions as $\sigma \circ \tau = \overline{\sigma} \cdot \tau$, and we call $\sigma \circ \tau$ the *composition* of σ and τ. That is, $\sigma \circ \tau$ is the function $\lambda\, x \in \mathcal{V}.\, \overline{\sigma}(\tau(x))$. This function is clearly a substitution (has finite support) since for all x outside $Supp(\sigma) \cup Supp(\tau)$ we have $\sigma \circ \tau(x) = x$, which proves that $\circ$ is indeed an operation. Further, a simple induction will show that $\overline{\sigma \circ \tau} = \overline{\sigma} \cdot \overline{\tau}$, which justifies naming this operation "composition". It is also straightforward to verify that the set of all substitutions forms a monoid under this operation, with the empty substitution serving as the identity element. We conclude:

Lemma A.1 *For any substitutions θ, σ, τ:*

(a) $\overline{\sigma \circ \tau} = \overline{\sigma} \cdot \overline{\tau}$;

(b) $\{\} \circ \theta = \theta \circ \{\} = \theta$;

(c) $\theta \circ (\sigma \circ \tau) = (\theta \circ \sigma) \circ \tau$.

In analogy with customary function composition, we define the powers of a substitution θ as $\theta^0 = \{\}, \theta^{i+1} = \theta \circ \theta^i$.

A substitution θ is called *idempotent* iff $\theta = \theta^2$, i.e., iff $\theta(x) = \overline{\theta}(\theta(x))$ for all $x \in \mathcal{V}$. Idempotent substitutions are nice because they yield a final result with only one application—once we have applied an idempotent substitution to a term, reapplying it will not give us anything new: $\overline{\theta}(\overline{\theta}(t)) = \overline{\theta}(t)$. Contrast this with a non-idempotent substitution such as $\sigma = \{x \mapsto f(y), y \mapsto a\}$. Applying σ to, say, $g(x)$, yields $g(f(y))$. But this can be viewed as an "incomplete" result in that it contains y, which σ maps to a. Thus if we apply σ once more we get another term, $g(f(a))$. This term can be seen as the final result in the sense of being a fixed point of $\overline{\sigma}$; because it does not contain any variables in the support of σ, if we apply σ to it again we will not get anything new. Thus we see that σ required two applications to give a final result. By contrast, idempotent substitutions reach a fixed point with just one application. The following result furnishes a simple mechanical test for detecting this property:

Lemma A.2 *θ is idempotent iff $RanVar(\theta) \cap Supp(\theta) = \emptyset$.*

A.2.2 Comparing substitutions

Let σ and τ be two substitutions and let $X \subseteq \mathcal{V}$. We say that σ and τ are *equal over* X, written $\sigma = \tau[X]$, iff $\sigma(x) = \tau(x)$ for all $x \in X$. When $X = \mathcal{V}$ we simply write $\sigma = \tau$. We say that σ is *less general than τ over X*, or equivalently, that τ is *more general than σ over X*, written $\sigma \leq \tau[X]$, iff there is a substitution θ such that for all $x \in X$, $\sigma(x)$ is a θ-instance of $\tau(x)$, i.e., such that $\sigma(x) = \overline{\theta}(\tau(x))$ for all $x \in X$. We can phrase this more succinctly as follows: $\sigma \leq \tau[X]$ iff there is a substitution θ such that $\sigma = \theta \circ \tau[X]$. Again, when $X = \mathcal{V}$ we simply write $\sigma \leq \tau$ and say that σ is less general than τ.

As an example, let $\sigma = \{x \mapsto f(g(a))\}$ and $\tau = \{x \mapsto f(y)\}$. Then $\sigma \leq \tau[\{x\}]$, i.e., τ is more general than σ over $\{x\}$, because $\sigma(x) = f(g(a))$ is an instance of $\tau(x) = f(y)$ under the substitution $\{y \mapsto g(a)\}$. However, τ is *not* more general than σ over $\{x, y\}$. To see this, observe that for any substitution θ for which $\sigma = \theta \circ \tau[\{x, y\}]$, we must have $\theta(y) = g(a)$. (Proof: If $\sigma = \theta \circ \tau[\{x, y\}]$ then $\sigma(x) = \theta \circ \tau(x)$, hence $\sigma(x) = \overline{\theta}(\tau(x))$, hence

$$f(g(a)) = \overline{\theta}(f(y)) = f(\overline{\theta}(y)) = f(\theta(y))$$

hence $\theta(y) = g(a)$.) But then $\theta \circ \tau(y) = \overline{\theta}(\tau(y)) = \overline{\theta}(y) = \theta(y) = g(a)$, whereas $\sigma(y) = y$. Thus $\sigma(y) \neq \theta \circ \tau(y)$, and consequently, $\sigma \neq \theta \circ \tau[\{x, y\}]$, contradicting the assumption $\sigma = \theta \circ \tau[\{x, y\}]$. In fact, by the same reasoning, τ fails to be more general than σ over *any* set of variables that includes both x and y.

The reader will verify that $\leq$ is a quasi-order:

Lemma A.3 *$\leq$ is a quasi-order. In particular, $\theta \leq \theta[X]$; and $\theta_1 \leq \theta_3[X]$ whenever $\theta_1 \leq \theta_2[X]$ and $\theta_2 \leq \theta_3[X]$.*

We define $\equiv$ as the symmetric closure of $\leq$: $\theta_1 \equiv \theta_2[X]$ iff $\theta_1 \leq \theta_2[X]$ and $\theta_2 \leq \theta_1[X]$. Clearly, $\equiv$ is an equivalence relation. Moreover, because $\theta_1(x)$ and $\theta_2(x)$ are instances of each other (for all $x \in X$), Theorem A.5 below tells us that $\theta_1(x)$ and $\theta_2(x)$ are literal variants. This means that θ_1 can be obtained from θ_2 via a renaming, and vice versa; i.e., if $\theta_1 \equiv \theta_2[X]$ then there are renamings σ, σ' such that $\theta_1 = \sigma \circ \theta_2$ and $\theta_2 = \sigma' \circ \theta_1$. (We cover renamings in the next section.) For this reason we may regard θ_1 and θ_2 as identical.

A.3 Patterns and matching

A term that contains variables can be viewed as a *pattern*. The variables act as place holders or "empty boxes" that may be filled with arbitrary terms, provided that the same term is consistently substituted for all different occurences of the same variable. Inserting terms in place of the variables yields *instances* of the pattern. Accordingly, this process is known as *instantiating* the pattern. For example, one instance of the term $plus(0, s(x))$ is $plus(0, s(times(0, y)))$, which is obtained by putting $times(0, y)$ in place of the variable x.

More formally, given two terms $s, t \in \mathbf{Terms}(\Sigma, \mathcal{V})$, we call s an *instance* of t (and in particular, a *ground instance* if s happens to be ground) iff there is a substitution θ such that $s = \overline{\theta}(t)$. We then write $s \trianglelefteq t$ and say that s *has the form of* t, or that it *matches* t under the substitution θ. The latter is called a *match* from t to s. We also say that t is *more general* than s, or that it *subsumes* s. For this reason, $\trianglelefteq$ is often called the *subsumption relation*.

Lemma A.4 *The subsumption relation is a quasi-order.*

Proof: Reflexivity follows immediately in virtue of the empty substitution. For transitivity, suppose that $s_1 \trianglelefteq s_2$ via σ and $s_2 \trianglelefteq s_3$ via τ. Then $\overline{\sigma \circ \tau}(s_3) = \overline{\sigma}(s_2) = s_1$, hence s_1 matches s_3 under the composition $\sigma \circ \tau$. ∎

Theorem A.5 below will show that the equivalence relation induced by $\trianglelefteq$ captures the identity of patterns. That is, if two terms are instances of each other then they represent the same pattern.

The subsumption relation is decidable. Specifically, there is an efficient algorithm for solving the following problem: given s and t, determine whether s is an instance of t, and if so, produce a matching substitution θ such that $s = \overline{\theta}(t)$. The algorithm in question is linear in the height of the pattern t, and this is what makes first-order pattern matching so expedient. Both matching and the related problem of unification are substantially more involved in the higher-order case.

A.3.1 Renamings

Consider the terms $s = f(x, g(y, x))$ and $t = f(z, g(w, z))$. It is clear that these two terms represent the same pattern. They only differ in the names of their variables: t uses z and w where s uses x and y, respectively. We say that s and t are *literal variants* of each other. This relationship is formalized via the notion of renaming substitutions.

A substitution is a *renaming* iff it is of the form

$$\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$$

where $\{x'_1, \ldots, x'_n\} = \{x_1, \ldots, x_n\}$; i.e., iff it is a permutation of its support, and by extension, a permutation of the entire set of variables $\mathcal{V}$. For example,

$$\theta_1 = \{x \mapsto y, y \mapsto z, z \mapsto x\}$$

is a renaming, but neither $\theta_2 = \{x \mapsto y\}$ nor $\theta_3 = \{x \mapsto y, y \mapsto z\}$ are. Intuitively, a substitution is a renaming iff applying it to an arbitrary term t will yield a term t' that represents the exact same pattern that t does. E.g., applying θ_1 above to $f(x, y, x)$ gives $f(y, z, y)$, which is the same pattern. But applying, say, θ_2 to $f(x, y)$ gives $f(y, y)$, which is a different pattern.

We now formally define a term t to be a *literal variant* (or simply a "variant") of a term s iff $t = \overline{\theta}(s)$ for some renaming θ. Observe that because every permutation has an inverse, this is a symmetric relation: if t is a literal variant of s then s is a literal variant of t . E.g. the inverse of $\theta_1 = \{x \mapsto y, y \mapsto z, z \mapsto x\}$ is $\theta_1^{-1} = \{x \mapsto z, y \mapsto x, z \mapsto y\}$. Thus, since $f(x, y, x)$ is a literal variant of $f(y, z, y)$ via θ_1, $f(y, z, y)$ is a literal variant of $f(x, y, x)$ via θ_1^{-1}. We will write $s \doteq t$ to indicate that s and t are literal variants. We call $\doteq$ the relation of "alphabetic equivalence" on Herbrand terms, and if $s \doteq t$, we say that s and t are the same "up to renaming".

As one would expect, $\doteq$ is an equivalence relation. It can be viewed as the equality relation on patterns. That is, if $s \doteq t$ then s and t represent the same pattern. We have already discussed symmetry. Reflexivity follows by virtue of the empty renaming $\{\}$; and transitivity from the fact that renamings can be composed (although this is not quite trivial to prove). However, a more informative way of showing $\doteq$ to be an equivalence relation is to point out that it is the symmetric closure of the subsumption relation, which we already know to be a quasi-order:

Theorem A.5 *Two terms are literal variants iff each is an instance of the other. In symbols, $s \doteq t$ iff $s \trianglelefteq t$ and $t \trianglelefteq s$.*

B
$\mathcal{CND}$

In this appendix we give rigorous definitions of some $\mathcal{CND}$ notions that were either omitted or only described informally in the body of the document.

B.1 Propositional $\mathcal{CND}$

We define $DDom(D)$, the *domain* of a deduction D, as follows:

$$\begin{array}{rcl}
DDom(P) & = & \{\,[]\} \\
DDom(Rule\ P_1, \ldots, P_n) & = & \{\,[]\} \cup \{\,[1], \ldots, [n]\} \\
DDom(\textbf{assume}\ P\ \textbf{in}\ D) & = & \{\,[]\} \cup \{\,[1]\} \cup \{\,2{::}p \mid p \in DDom(D)\} \\
DDom(D_1; D_2) & = & \{\,[]\} \cup \{\,1{::}p \mid p \in DDom(D_1)\} \cup \{\,2{::}p \mid p \in DDom(D_2)\}
\end{array}$$

We can now define a function $DLabel$ that takes a deduction D and a "position" $p \in DDom(D)$ and returns whatever part of D appears there:

$$\begin{array}{rcl}
DLabel(P, []) & = & P \\
DLabel(Prim\text{-}Rule\ P_1, \ldots, P_n, []) & = & Prim\text{-}Rule \\
DLabel(Prim\text{-}Rule\ P_1, \ldots, P_n, [i]) & = & P_i \text{ (for } i = 1, \ldots, n) \\
DLabel(\textbf{assume}\ P\ \textbf{in}\ D, []) & = & \textbf{assume} \\
DLabel(\textbf{assume}\ P\ \textbf{in}\ D, [1]) & = & P \\
DLabel(\textbf{assume}\ P\ \textbf{in}\ D, 2{::}p) & = & DLabel(D, p) \\
DLabel(D_1; D_2, []) & = & ;
\end{array}$$

$$\begin{array}{rcl} DLabel(D_1; D_2, 1{::}p) & = & DLabel(D_1, p) \\ DLabel(D_1; D_2, 2{::}p) & = & DLabel(D_2, p) \end{array}$$

We say that a deduction D' *occurs in* D *at* some $p \in DDom(D)$ iff $DLabel(D', q) = DLabel(D, p \oplus q)$ for every $q \in DDom(D')$. By a *subdeduction* of D we will mean any deduction that occurs in D at some position. We define a *thread* of D as any subdeduction of D of the form $D_1; D_2; \ldots; D_n; D_{n+1}$, $n \geq 1$.[1] We call $D_1, \ldots, D_n$ the *elements* of the thread. For each $i = 1, \ldots, n$, we say that D_i *dominates* every D_j, $i < j \leq n + 1$; and we call $D_1, \ldots, D_n$ the *dominating elements* of the thread. In general, we say that a deduction D' dominates a deduction D'' in D iff there is a position $p \in DDom(D)$ and a non-empty list $q = [2, \ldots, 2]$ (i.e., q is a list of one or more 2s) such that

1. $DLabel(D, p) = ;$
2. D' occurs in D at $p \oplus [1]$
3. $DLabel(D, p \oplus q') = ;$ for every prefix $q' \sqsubseteq q$
4. D'' occurs in D at $p \oplus q \oplus [1]$.

Clearly, the dominance relation imposes a total ordering on the elements of a thread.

Next, let $p \in DDom(D)$. We will say that p is a *trailing position* (or *conclusion position*) in D iff there is no $q \in DDom(D)$ such that $DLabel(D, q) =;$ and $p = q \oplus [1]$. It follows that a subdeduction of D occurs in a *non*-trailing position in D iff it is a dominating element of some thread of D.

Recall from Sec. 4.2 that a $\mathcal{CND}$ deduction is well-formed iff every primitive subdeduction of it has the right number and form of arguments. Here this is formally defined via rules that establish judgements of the form $\vdash_W D$, asserting that D is well-formed. For primitive deductions we have the following axiom schemas:

$$\frac{}{\vdash_W \textbf{modus-ponens}\ P \Rightarrow Q, P} \qquad \frac{}{\vdash_W \textbf{modus-tollens}\ P \Rightarrow Q, \neg Q}$$

$$\frac{}{\vdash_W \textbf{both}\ P, Q} \qquad \frac{}{\vdash_W \textbf{left-and}\ P \wedge Q} \qquad \frac{}{\vdash_W \textbf{right-and}\ P \wedge Q}$$

$$\frac{}{\vdash_W \textbf{left-either}\ P, Q} \qquad \frac{}{\vdash_W \textbf{right-either}\ P, Q}$$

$$\frac{}{\vdash_W \textbf{constructive-dilemma}\ P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q}$$

[1] Or more precisely, recalling that composition associates to the right, as any subdeduction of the form $D_1; (D_2; \cdots; (D_n; D_{n+1}) \cdots)$.

$$\frac{}{\vdash_W \textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P} \qquad \frac{}{\vdash_W \textbf{absurd } P, \neg P}$$

$$\frac{}{\vdash_W \textbf{double-negation } \neg\neg P} \qquad \frac{}{\vdash_W \textbf{left-iff } P \Leftrightarrow Q} \qquad \frac{}{\vdash_W \textbf{right-iff } P \Leftrightarrow Q}$$

For non-primitive deductions we have:

$$\frac{}{\vdash_W P} \qquad \frac{\vdash_W D}{\vdash_W \textbf{assume } P \textbf{ in } D} \qquad \frac{\vdash_W D_1 \quad \vdash_W D_2}{\vdash_W D_1; D_2}$$

The *conclusion* $\mathcal{C}(D)$ of a well-formed primitive deduction D is defined as:

$$\begin{array}{rcl}
\mathcal{C}(\textbf{modus-ponens } P \Rightarrow Q, P) & = & Q \\
\mathcal{C}(\textbf{modus-tollens } P \Rightarrow Q, \neg Q) & = & \neg P \\
\mathcal{C}(\textbf{double-negation } \neg\neg P) & = & P \\
\mathcal{C}(\textbf{both } P, Q) & = & P \wedge Q \\
\mathcal{C}(\textbf{left-and } P \wedge Q) & = & P \\
\mathcal{C}(\textbf{right-and } P \wedge Q) & = & Q \\
\mathcal{C}(\textbf{left-either } P, Q) & = & P \vee Q \\
\mathcal{C}(\textbf{right-either } P, Q) & = & P \vee Q \\
\mathcal{C}(\textbf{constructive-dilemma } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q) & = & Q \\
\mathcal{C}(\textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P) & = & P \Leftrightarrow Q \\
\mathcal{C}(\textbf{left-iff } P \Leftrightarrow Q) & = & P \Rightarrow Q \\
\mathcal{C}(\textbf{right-iff } P \Leftrightarrow Q) & = & Q \Rightarrow P \\
\mathcal{C}(\textbf{absurd } P, \neg P) & = & \textbf{false}
\end{array}$$

For non-primitive deductions we have:

$$\begin{array}{rcl}
\mathcal{C}(P) & = & P \\
\mathcal{C}(\textbf{assume } P \textbf{ in } D) & = & P \Rightarrow \mathcal{C}(D) \\
\mathcal{C}(D_1; D_2) & = & \mathcal{C}(D_2)
\end{array}$$

Finally, we define the auxiliary function *do-prim-rule* used by the interpeter *Eval* (Fig. 4.4) as *do-prim-rule* $= f$, where

$f(\textbf{modus-ponens}, [P \Rightarrow Q, P], \beta \cup \{P \Rightarrow Q, P\}) = Q$
$f(\textbf{modus-tollens}, [P \Rightarrow Q, \neg Q], \beta \cup \{P \Rightarrow Q, \neg Q\}) = \neg P$
$f(\textbf{double-negation}, [\neg\neg P], \beta \cup \{\neg\neg P\}) = P$
$f(\textbf{both}, [P, Q], \beta \cup \{P, Q\}) = P \wedge Q$
$f(\textbf{left-and}, [P \wedge Q], \beta \cup \{P \wedge Q\}) = P$
$f(\textbf{right-and}, [P \wedge Q], \beta \cup \{P \wedge Q\}) = Q$
$f(\textbf{constructive-dilemma}, [P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q], \beta \cup \{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\}) = Q$

$f(\textbf{left-either}, [P, Q], \beta \cup \{P\}) = P \vee Q$
$f(\textbf{right-either}, [P, Q], \beta \cup \{Q\}) = P \vee Q$
$f(\textbf{equivalence}, [P \Rightarrow Q, Q \Rightarrow P], \beta \cup \{P \Rightarrow Q, Q \Rightarrow P\}) = P \Leftrightarrow Q$
$f(\textbf{left-iff}, [P \Leftrightarrow Q], \beta \cup \{P \Leftrightarrow Q\}) = P \Rightarrow Q$
$f(\textbf{right-iff}, [P \Leftrightarrow Q], \beta \cup \{P \Leftrightarrow Q\}) = Q \Rightarrow P$
$f(\textbf{absurd}, [P, \neg P], \beta \cup \{P, \neg P\}) = \textbf{false}$
$f(\_, \_, \_) = \mathit{error}$

The last clause dictates that an error must occur if the list of arguments supplied to f does not match any of the preceding patterns.

B.2 Predicate $\mathcal{CND}$

B.2.1 Formulas

For any formula F, we define $\mathit{FDom}(F)$—the *domain* of F—as a set of positions (lists of positive integers), similarly to the definition of TDom in Appendix A:

$$\begin{array}{rcl}
\mathit{FDom}(R(t_1,\ldots,t_n)) & = & \{[]\} \cup \{i{::}p \mid p \in \mathit{TDom}(t_i), i = 1,\ldots,n\} \\
\mathit{FDom}(\neg F) & = & \{[]\} \cup \{1{::}p \mid p \in \mathit{FDom}(F)\} \\
\mathit{FDom}(F \circ G) & = & \{[]\} \cup \{1{::}p \mid p \in \mathit{FDom}(F)\} \cup \{2{::}p \mid p \in \mathit{FDom}(G)\} \\
\mathit{FDom}((Q\,x)\,F) & = & \{[], [1]\} \cup \{2{::}p \mid p \in \mathit{FDom}(F)\}
\end{array}$$

for $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, $Q \in \{\forall, \exists\}$. Continuing the analogy, we define a binary function FLabel that takes a formula F and a position $p \in \mathit{FDom}(F)$ and returns whatever datum occurs there:

$$\begin{array}{rcl}
\mathit{FLabel}(R(t_1,\ldots,t_n), []) & = & R \\
\mathit{FLabel}(R(t_1,\ldots,t_n), i{::}p) & = & \mathit{TLabel}(t_i, p) \\
\mathit{FLabel}(\neg F, []) & = & \neg \\
\mathit{FLabel}(\neg F, 1{::}p) & = & \mathit{FLabel}(F, p) \\
\mathit{FLabel}(F \circ G, []) & = & \circ \\
\mathit{FLabel}(F \circ G, 1{::}p) & = & \mathit{FLabel}(F, p) \\
\mathit{FLabel}(F \circ G, 2{::}p) & = & \mathit{FLabel}(G, p) \\
\mathit{FLabel}((Q\,x)\,F, []) & = & Q \\
\mathit{FLabel}((Q\,x)\,F, [1]) & = & x \\
\mathit{FLabel}((Q\,x)\,F, 2{::}p) & = & \mathit{FLabel}(F, p)
\end{array}$$

An *occurence* of a term t in a formula F is a position $p \in \mathit{FDom}(F)$ such that $\mathit{TLabel}(t, q) = \mathit{FLabel}(F, p \oplus q)$ for all $q \in \mathit{TLabel}(t)$. If such an occurence exists we say that t *occurs* in F (at position p). Likewise, by an occurence of a formula G in F we will mean a position $p \in FDom(F)$ such that

$$FLabel(G, q) = FLabel(F, p \oplus q)$$

for all $q \in FLabel(G)$; and we say that G occurs in F at p to mean that p is such an occurence. A *subformula* of F is any formula that occurs in F at some position.

A variable x *occurs bound* in F at position q iff there is a prefix $p \sqsubset q$ and a quantifier Q such that $FLabel(F, p) = Q$ and $FLabel(F, p \oplus [1]) = x$; x occurs *free* in F at q iff it does not occur bound in F at q. By a bound (free) occurence of x in F we will mean a position $p \in FDom(F)$ such that x occurs bound (free) in F at p. We write $BV(F)$ ($FV(F)$) for the set of all variables that occur bound (free) in F at some position, and we set $Var(F) = BV(F) \cup FV(F)$. The definitions of these three sets are readily generalized to arbitrary sets of formulas; e.g., for any $\Phi \subseteq \mathbf{Form}(\boldsymbol{\Omega}, \mathcal{V})$ we set

$$FV(\Phi) = \bigcup_{F \in \Phi} FV(F)$$

and likewise for $BV(\Phi)$ and $Var(\Phi)$.

B.2.2 Deductions

The *domain* of a deduction D, denoted $DDom(D)$, is defined as a set of lists of positive integers ("positions"), in a manner similar to the definition of $TDom$ and $FDom$:

$$\begin{array}{rcl}
DDom(F) & = & FDom(F) \\
DDom(\textit{Prim-Rule}\ F_1, \ldots, F_n) & = & \{[]\} \cup \{i{::}p \mid p \in FDom(F_i), i = 1, \ldots, n\} \\
DDom(\textbf{specialize}\ F\ \textbf{with}\ t) & = & \{[]\} \cup \{1{::}p \mid p \in FDom(F)\} \cup \{2{::}p \mid p \in TDom(t)\} \\
DDom(\textbf{ex-generalize}\ F\ \textbf{from}\ t) & = & \{[]\} \cup \{1{::}p \mid p \in FDom(F)\} \cup \{2{::}p \mid p \in TDom(t)\} \\
DDom(\textbf{assume}\ F\ \textbf{in}\ D) & = & \{[]\} \cup \{1{::}p \mid p \in FDom(F)\} \cup \{2{::}p \mid p \in DDom(D)\} \\
DDom(D_1; D_2) & = & \{[]\} \cup \{1{::}p \mid p \in DDom(D_1)\} \cup \{2{::}p \mid p \in DDom(D_2)\} \\
DDom(\textbf{pick-any}\ x\ \textbf{in}\ D) & = & \{[], [1]\} \cup \{2{::}p \mid p \in DDom(D)\} \\
DDom(\textbf{pick-witness}\ x\ \textbf{for}\ F\ \textbf{in}\ D) & = & \{[], [1]\} \cup \{2{::}p \mid p \in FDom(F) \cup \{3{::}p \mid p \in DDom(D)\}
\end{array}$$

Next we define a function $DLabel$ that does for deductions what $TLabel$ and $FLabel$ do for terms and formulas, respectively:

$$\begin{array}{rcl}
DLabel(F, p) & = & FLabel(F, p) \\
DLabel(\textit{Prim-Rule}\ F_1, \ldots, F_n, []) & = & \textit{Prim-Rule} \\
DLabel(\textit{Prim-Rule}\ F_1, \ldots, F_n, i{::}p) & = & FLabel(F_i, p) \\
DLabel(\textbf{specialize}\ F\ \textbf{with}\ t, []) & = & \textbf{specialize} \\
DLabel(\textbf{specialize}\ F\ \textbf{with}\ t, 1{::}p) & = & FLabel(F, p)
\end{array}$$

$$
\begin{array}{rcl}
\mathit{DLabel}(\textbf{specialize } F \textbf{ with } t, 2{::}p) & = & \mathit{TLabel}(t,p) \\
\mathit{DLabel}(\textbf{ex-generalize } F \textbf{ from } t, []) & = & \textbf{ex-generalize} \\
\mathit{DLabel}(\textbf{ex-generalize } F \textbf{ from } t, 1{::}p) & = & \mathit{FLabel}(F,p) \\
\mathit{DLabel}(\textbf{ex-generalize } F \textbf{ from } t, 2{::}p) & = & \mathit{TLabel}(t,p) \\
\mathit{DLabel}(\textbf{assume } F \textbf{ in } D, []) & = & \textbf{assume} \\
\mathit{DLabel}(\textbf{assume } F \textbf{ in } D, 1{::}p) & = & \mathit{FLabel}(F,p) \\
\mathit{DLabel}(\textbf{assume } F \textbf{ in } D, 2{::}p) & = & \mathit{DLabel}(D,p) \\
\mathit{DLabel}(D_1; D_2, []) & = & ; \\
\mathit{DLabel}(D_1; D_2, 1{::}p) & = & \mathit{DLabel}(D_1,p) \\
\mathit{DLabel}(D_1; D_2, 2{::}p) & = & \mathit{DLabel}(D_2,p) \\
\mathit{DLabel}(\textbf{pick-any } x \textbf{ in } D, []) & = & \textbf{pick-any} \\
\mathit{DLabel}(\textbf{pick-any } x \textbf{ in } D, [1]) & = & x \\
\mathit{DLabel}(\textbf{pick-any } x \textbf{ in } D, 2{::}p) & = & \mathit{DLabel}(D,p) \\
\mathit{DLabel}(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D, []) & = & \textbf{pick-witness} \\
\mathit{DLabel}(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D, [1]) & = & x \\
\mathit{DLabel}(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D, 2{::}p) & = & \mathit{FLabel}(F,p) \\
\mathit{DLabel}(\textbf{pick-witness } x \textbf{ for } F \textbf{ in } D, 3{::}p) & = & \mathit{DLabel}(D,p)
\end{array}
$$

An *occurence* of a term t in a deduction D is a position $p \in \mathit{DDom}(D)$ such that $\mathit{TLabel}(t,q) = \mathit{DLabel}(D, p \oplus q)$ for all $q \in \mathit{TDom}(t)$; if such an occurence exists we say that t *occurs* in F (at position p). Likewise, by an occurence of a formula F in D we will mean a position $p \in \mathit{DDom}(D)$ such that $\mathit{FLabel}(F,q) = \mathit{DLabel}(D, p \oplus q)$ for all $q \in \mathit{FDom}(F)$; and we say that F occurs in D at p to mean that p is such an occurence. Finally, an occurence of a deduction D' in D is a position $p \in \mathit{DDom}(D)$ such that $\mathit{DLabel}(D', q) = \mathit{DLabel}(D, p \oplus q)$ for all $q \in \mathit{DDom}(D')$. If such an occurence exists we say that D' occurs in D, and we refer to D' as a *subdeduction* of D. The *size* of a deduction D, denoted $\mathit{SZ}(D)$, is defined as the cardinality of the set $\mathit{DDom}(D)$.

There are two ways for a variable occurence to be bound in a first-order $\mathcal{CND}$ deduction: through a quantifier, or through **pick-any** or **pick-witness**. Specifically, an occurence q of a variable x in D is *quantifier-bound* iff there is a prefix $p \sqsubset q$ and a quantifier Q such that $\mathit{DLabel}(D,p) = Q$ and $\mathit{DLabel}(D, p \oplus [1]) = x$. An occurence of x that is not quantifier-bound is said to be an *eigenvariable occurence* (or "eigenbound") iff there is a prefix $p \sqsubset q$ such that (a) $\mathit{DLabel}(D,p) =$ **pick-any** or $\mathit{DLabel}(D,p) =$ **pick-witness**; and (b) $\mathit{DLabel}(D, p \oplus [1]) = x$. A *free* occurence of x in D is one that is neither quantifier-bound nor eigenbound. We write $FV(D)$ and $EV(D)$ for the sets of variables that have free and eigenbound occurences in D, respectively; $\mathit{Var}(D)$ denotes the set of variables that occur in D.

Well-formed deductions in predicate $\mathcal{CND}$ are defined as in the propositional case: via a collection of similar rules that establish judgements of the form $\vdash_W D$:

$$\frac{}{\vdash_W \textbf{modus-ponens } F \Rightarrow G, F} \qquad \frac{}{\vdash_W \textbf{modus-tollens } F \Rightarrow G, \neg G}$$

$$\frac{}{\vdash_W \textbf{both } F, G} \qquad \frac{}{\vdash_W \textbf{left-and } F \wedge G} \qquad \frac{}{\vdash_W \textbf{right-and } F \wedge G}$$

$$\frac{}{\vdash_W \textbf{left-either } F, G} \qquad \frac{}{\vdash_W \textbf{right-either } F, G}$$

$$\frac{}{\vdash_W \textbf{constructive-dilemma } F_1 \vee F_2, F_1 \Rightarrow G, F_2 \Rightarrow G}$$

$$\frac{}{\vdash_W \textbf{equivalence } F \Rightarrow G, G \Rightarrow F} \qquad \frac{}{\vdash_W \textbf{absurd } F, \neg F}$$

$$\frac{}{\vdash_W \textbf{double-negation } \neg\neg F} \qquad \frac{}{\vdash_W \textbf{left-iff } F \Leftrightarrow G} \qquad \frac{}{\vdash_W \textbf{right-iff } F \Leftrightarrow G}$$

$$\frac{}{\vdash_W \textbf{specialize } (\forall\, x)\, F \textbf{ with } t} \qquad \frac{}{\vdash_W \textbf{ex-generalize } (\exists\, x)\, F \textbf{ from } t}$$

For non-primitive deductions we have:

$$\frac{}{\vdash_W F} \qquad \frac{\vdash_W D}{\vdash_W \textbf{assume } F \textbf{ in } D} \qquad \frac{\vdash_W D_1 \quad \vdash_W D_2}{\vdash_W D_1; D_2}$$

$$\frac{\vdash_W D}{\vdash_W \textbf{pick-any } x \textbf{ in } D} \qquad \frac{\vdash_W D}{\vdash_W \textbf{pick-witness } x \textbf{ for } (\exists\, x)\, F \textbf{ in } D}$$

The function *do-prim-rule* used by the interpeter *Eval* shown in Fig. 6.6 is defined just as in the propositional case (Section B.1).

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[2] M. Aigner and G. M. Ziegler. *Proofs from the book*. Springer, 1998.

[3] K. Arkoudas. Athena: a formal system integrating deduction and computation. Forthcoming, at `www.ai.mit.edu/projects/express`.

[4] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North Holland, 1984.

[5] M. Bergmann, J. Moor, and J. Nelson. *The Logic Book*. Random House, New York, 1980.

[6] E. W. Beth. Semantic entailment and formal derivability. *Mededlingen der Koninklijke Nederlandse Akademie van Wetenschappen*, 18(13):309–342, 1955.

[7] E. W. Beth. *The foundations of mathematics*. North Holland, 1959.

[8] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press, New York, 1988.

[9] N. G. De Brujin. The Automath checking project. In P. Braffort, editor, *Proceedings of Symposium on APL*, Paris, France, December 1973.

[10] N. G. De Brujin. A survey of the project Automath. In J. Hindley and J. R. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalisms*. Academic Press, 1980.

[11] N. G. De Brujin. Type-theoretical checking and philosophy of mathematics. In G. Sambin and J. Smith, editors, *Twenty five years of constructive type theory*, volume 36 of *Oxford Logic Guides*. Oxford University Press, 1998.

[12] S. Buss. Introduction to proof theory. In S. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1998.

[13] R. Carnap. *Logical Foundations of Probability*. University of Chicago Press, 1950.

[14] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[15] D. Clément, J. Despeyroux, T. Desperyoux, and G. Kahn. Natural semantics on the computer. Research Report RR 416, INRIA, Sophia-Antipolis, France, June 1985.

[16] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, EngleWood Cliffs, New Jersey, 1986.

[17] I. M. Copi. *Symbolic Logic*. Macmillan Publishing Co., New York, 5th edition, 1979.

[18] T. Coquand. Metamathetical investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, London, 1990.

[19] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[20] J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In R. E. A. Mason, editor, *Information Processing*, volume 83, pages 909–914. Elsevier, Amsterdam, Holland, 1983.

[21] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[22] D. V. Dalen. *Logic and Structure*. Springer Verlag, 1983.

[23] A. G. Dragalin. *Mathematical Intuitionism. Introduction to Proof Theory*, volume 67 of *Translations of Mathematical Monographs*. American Mathematical Society, Providence, RI, 1988.

[24] M. Dummett. *Frege: Philosophy of Language*. Harper & Row, 1973.

[25] F. B. Fitch. *Symbolic Logic: an Introduction*. The Ronald Press Co., New York, 1952.

[26] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proceedings of Symposium in Applied Mathematics*, pages 19–32. AMS, 1967.

[27] J. H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[28] S. J. Garland and J. V. Guttag. A guide to LP, The Larch Prover. Research Report 82, Systems Research Center, DEC, 130 Lytton Avenue, Palo Alto, California 94301, December 1991.

[29] G. Gentzen. *The collected papers of Gerhard Gentzen*. North-Holland, Amsterdam, Holland, 1969. English translations of Gentzen's papers, edited and introduced by M. E. Szabo.

[30] D. Gifford and F. Turbak. Applied semantics of programming languages. Unpublished book draft used in MIT course 6.821.

[31] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[32] M. J. C. Gordon. *The denotational description of programming languages*. Springer-Verlag, 1979.

[33] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.

[34] J. Goubault-Larrecq and I. Mackie. *Proof Theory and Automated Deduction*. Kluwer Academic Publishers, 1997.

[35] J. Guttag and B. Liskov. *Abstraction and specification in program development*. MIT Press, 1986.

[36] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[37] J. Herbrand. Sur la Théorie de la Démonstration. In W. Goldfarb, editor, *Logical Writings of Herbrand*. Cambridge University Press, 1971.

[38] C. A. R. Hoare. An axiomatic basis for computer programming. *Acta Informatica*, 1:271–281, 1972.

[39] W. Hodges. Elementary predicate logic. In D. M. Gabbay and F. Guenthner, editors, *Elements of Classical Logic*, volume 1 of *Handbook of Philosophical Logic*. D. Reidel Publishing Company, 1983.

[40] G. Kahn. Natural semantics. In *Proceedings of Theoretical Aspects of Computer Science*, Passau, Germany, February 1987.

[41] D. Kalish and R. Montague. *Logic: Techniques of Formal Reasoning*. Harcourt Brace Jovanovich, Inc., New York, 1964. Second edition in 1980, with G. Mar.

[42] S. C. Kleene. *Mathematical Logic*. John Wiley & Sons, 1967.

[43] M. Kline. *Mathematics: The Loss of Certainty*. Oxford University Press, 1982.

[44] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[45] L. Lamport. How to write a proof. Research Report 94, Systems Research Center, DEC, February 1993.

[46] E. J. Lemmon. *Beginning Logic*. Hackett Publishing Company, 1978.

[47] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Computer Science Series, 1974.

[48] A. Martelli and U. Montanari. Unification in linear time and space: A structured presentation. Technical Report B76-16, University of Pisa, 1976.

[49] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[50] S. Jáskowski. On the rules of suppositions in formal logic. *Studia Logica*, 1, 1934.

[51] D. McAllester. *Ontic*. MIT Press, 1989.

[52] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[53] G. Necula and P. Lee. Efficient representation and validation of logical proofs. Computer Science Technical Report CMU-CS-97-172, CMU, October 1997.

[54] S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language (draft). Research report, Computer Science Laboratory, SRI International, Menlo Park, California, February 1993.

[55] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Int. Conf. Log. Prog. Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[56] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[57] L. Paulson. *Isabelle, A Generic Theorem Prover.* Lecture Notes in Computer Science. Springer-Verlag, 1994.

[58] G. D. Plotkin. A structural approach to operational semantics. Research Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[59] D. Prawitz. *Natural Deduction.* Almqvist & Wiksell, Stockhol, Sweden, 1965.

[60] W. V. Quine. *Methods of Logic.* Harvard University Press, 4th edition, 1982.

[61] N. Rescher. *Introduction to Logic.* St. Martin's Press, 1964.

[62] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the 1999 Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[63] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[64] P. Rudnicki. An overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Chalmers University of Technology, Bastad, 1992.

[65] D. A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[66] J. E. Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory.* MIT Press, 1981.

[67] P. Suppes. *Introduction to Logic.* Van Nostrand, Princeton, NJ, 1957.

[68] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, Cambridge, England, 1996.

[69] W. Wechler. *Universal Algebra for Computer Scientists.* Springer-Verlag, 1992.

[70] J. Welsh, W. J. Sneeringer, and C. A. R. Hoare. Ambiguities and Insecurities in PASCAL. *Software Practice and Experience*, 7, 1977.