Certified Computation

Konstantine Arkoudas

April 4, 2001

Abstract

This paper introduces the notion of certified computation. A certified computation does not only produce a result r, but also a *correctness certificate*, which is a formal proof that r is correct. This can greatly enhance the credibility of the result: if we trust the axioms and inference rules that are used in the certificate, then we can be assured that r is correct. In effect, we obtain a *trust reduction*: we no longer have to trust the entire computation; we only have to trust the certificate. Typically, the reasoning used in the certificate is much simpler and easier to trust than the entire computation.

Certified computation has two main applications: as a software engineering discipline, it can be used to increase the reliability of our code; and as a framework for cooperative computation, it can be used whenever a code consumer executes an algorithm obtained from an untrusted agent and needs to be convinced that the generated results are correct.

We propose DPLs (Denotational Proof Languages) as a uniform platform for certified computation. DPLs enforce a sharp separation between logic and control and offer versatile mechanicms for constructing certificates. We use Athena as a concrete DPL to illustrate our ideas, and we present two examples of certified computation, giving full working code in both cases.

1.1 Introduction

1.1.1 Algorithms, soundness, and completeness

In general terms, conventional computation may be viewed as follows: we start out with an abstract relation of interest

$$R \subseteq A_1 \times \cdots \times A_n \times B$$

where the A_i are the "input domains" and B is the "output domain" of the problem. Our task is to devise an algorithm f_R that will take any given vector of elements $x_1, \ldots, x_n$ as input, with each x_i drawn from the domain A_i; and will produce an element $y \in B$ such that $R(x_1, \ldots, x_n, y)$, if such a y exists at all. Usually R is a function, and oftentimes it is a *total* function as well, in which case the task of the algorithm f_R is simply to compute R: to produce the unique y such that $R(x_1, \ldots, x_n, y)$. For instance, R might be the greatest common divisor (gcd) function, with two input domains, A_1 and A_2, the integers, which also comprise the output domain B, and in that case f_R might be Euclid's algorithm, a procedure that takes two numbers and produces their gcd. Or the input domain A_1 might be the set of all C programs, the output domain might be the set of all machine-language programs for a certain computer architecture, and R might relate a C program to a machine language program iff the two are semantically equivalent. The desired f_R algorithm will then be a C compiler for the computer at hand. Note that in this case R is not a function; for any given input C program there will be many machine language programs with the same behavior. All we want is to find some such program.[1]

For some inputs $x_1, \ldots, x_n$ our algorithm f_R might get into an infinite loop or generate an error. In those cases we say that f_R "diverges" on $x_1, \ldots, x_n$, and we indicate this by

[1] Of course in practice we want to maximize the efficiency of the translated program.

writing $f_R(x_1,\ldots,x_n)\uparrow$. On the other hand, if f_R successfully produces an object $y \in B$ for given $x_1,\ldots,x_n$, we write $f_R(x_1,\ldots,x_n)\downarrow$ and say that f_R "converges" on $x_1,\ldots,x_n$.

Now the soundness problem is this: we have n inputs $x_1,\ldots,x_n$. We feed them to the algorithm f_R. The algorithm computes away and eventually comes back with a result y. How do we know that y is a correct result? That is, how do we know that we indeed have $R(x_1,\ldots,x_n,y)$? The answer is simple: we don't. We either have to take output correctness for granted, or else we have to verify it ourselves, usually by testing it empirically. For instance, if our gcd algorithm yields the result 26 for the inputs 442 and 1014, then we might try to manually check that 26 divides both 442 and 1014, and that no other number between 26 and 442 has this property. Or if our C compiler translates a given program P_C into a machine language program P_M, then we might try running P_C and P_M on various inputs to make sure that they behave identically. This is known as "testing", or "debugging". As we will explain shortly, this process has several shortcomings.

Before we proceed, let us make these ideas more precise. Let us say that our algorithm f_R is *partially correct*, or *sound*, iff

$$R(x_1,\ldots,x_n,f_R(x_1,\ldots,x_n))$$

whenever $f_R(x_1,\ldots,x_n)\downarrow$. In words, f_R is sound iff it never produces any incorrect results. Equivalently: whenever f_R produces a result $y \in B$ for some inputs $x_1,\ldots,x_n$, that result is correct, meaning that we have $R(x_1,\ldots,x_n,y)$.

Conversely, we will say that f_R is *complete* iff $f_R(x_1,\ldots,x_n)\downarrow$ whenever

$$(\exists\, y)\, R(x_1,\ldots,x_n,y).$$

That is, if there exists an answer for a given input list $x_1,\ldots,x_n$, then the algorithm converges on $x_1,\ldots,x_n$, meaning that it produces *some* result. That result might or might not be correct—that is a soundness issue. But there will be a result; the algorithm will not loop forever, nor will it halt in error.

Neither soundness nor completeness in isolation suffices to guarantee that our algorithm works properly. Consider, for instance, an algorithm that always gets into an infinite loop, no matter what the inputs are. Such an algorithm is trivially sound: it never produces an incorrect result, simply because it never produces any result at all. Conversely, consider an algorithm that always outputs a fixed element $y \in B$, again regardless of the inputs. Such an algorithm is trivially complete. It always yields an answer, but in most cases that will be the wrong answer. To ensure that our program works properly, we must have both properties. Accordingly, we say that an algorithm is *totally correct*, or simply "correct", iff it is both sound and complete.

In this discussion we will only be concerned with algorithm soundness, i.e., with partial correctness. Of course that is only half of the picture. As we explained above, for example, any algorithm that always gets into an infinite loop or raises an exception is trivially sound, but useless. Nevertheless, soundness is an important part of the picture, much more important than the foregoing pathological scenario might suggest. Infinite loops and exceptions do occur, but they are relatively rare. Most traditional computations actually produce *some* result. The problem is that usually we have no idea whether that result is legitimate. It would clearly be of significant practical value if we had a guarantee that the computed result is in fact correct.

1.1.2 Testing and verification

How might we get such a guarantee? As we mentioned earlier, one alternative that suggests itself is testing: we write a program T_R, call it "the tester", that takes the inputs $x_1, \ldots, x_n$ and the output y and determines whether or not $R(x_1, \ldots, x_n, y)$. Then we transform our program from a simple top-level call $f_R(x_1, \ldots, x_n)$ to this:

$$\begin{array}{l} \textbf{let } \mathit{result} = f_R(x_1, \ldots, x_n); \\ \textbf{let } \mathit{result\text{-}holds?} = T_R(x_1, \ldots, x_n, \mathit{result}); \\ \textbf{if } \mathit{result\text{-}holds?} \textbf{ then} \\ \quad \text{output } \mathit{result} \\ \textbf{else} \\ \quad \text{error("Incorrect result.").} \end{array} \tag{1.1}$$

For instance, if f_R is a sorting algorithm then the tester T_R would be a function that takes two arrays and returns `true` if the second is a sorted permutation of the first, and `false` otherwise. That is roughly the idea behind enforcing invariants in Eiffel [8], or using assertions in C and C++ [12].

Clearly, this scheme does offer soundness: if 1.1 produces a result, then we can rest assured that the result is correct—or can we? The answer is a qualified yes: we can be assured that the result is correct *only to the extent that we trust* T_R. Unfortunately, the tester T_R is often complicated and difficult to implement, so our faith in it might well turn out to be almost as weak as our faith in f_R, or perhaps even weaker. Even for something as simple as sorting, testers are not straightforward to write. A corollary of this problem is inefficiency: because the tester can be quite involved, its computational overhead imposes a significant efficiency penalty on the overall program 1.1. Indeed, in practice even light-weight assertions and invariants (e.g. boolean expressions that only check that certain input pointers are not null or that an output number is positive) are only used during development. All but the most rudimentary sanity checks are typically removed from the final code.

Another drawback of testing is that it offers little insight into *why* a result is incorrect. Where does f_R go wrong? The tester will not give us any clue; it will simply pontificate that the result is wrong. By contrast, we will see that in a DPL the result is certified *as it is generated*, so if we do something wrong the proof will break down at the exact point of the problem, and the DPL run-time environment will issue a useful and informative error message.

But perhaps the most important drawback of testing is that it might be more than just difficult—it might well be downright *impossible.* It is folk wisdom in Computer Science that verifying a solution is easier than finding one, so one would expect "the tester" T_R to be easier than "the finder" f_R. But in fact oftentimes verification is more difficult: it can be easier to find a solution than to check it! A prime example of this arises in compilation: although it is possible (and even relatively inexpensive) to mechanically *find* a machine-language program that simulates a given program in the source language, it is impossible to mechanically *check* that a machine-language program respects the semantics of a source-language program. No such tester exists. The best we can do is randomly probe the two programs at various points in their input space and verify that they behave identically *at those points.* The degree of confidence we will attain by doing this is commensurate to the extensiveness of our testing, but it will never be perfect. At no point will we be able to conclude for sure that the result is correct. As another example, consider the execution of a Prolog query. To be certain that a positive answer from our Prolog system is correct we must be certain that the (properly instantiated) query follows logically from the clauses of our program. Determining this with certainty is impossible in the general case.

In general, a tester T_R is really nothing but a decision procedure for R: we give it $x_1, \ldots, x_n, y$ and ask it to tell us whether or not $R(x_1, \ldots, x_n, y)$, i.e., whether y is a correct solution for the inputs $x_1, \ldots, x_n$. For many interesting problems the relation R is undecidable, and hence there is no tester for it. In those cases testing simply cannot offer us any correctness guarantee.

Another approach is traditional program verification, whereby we *prove* once and for all that the algorithm will *never* produce incorrect results. In principle this is clearly a superior approach, but in practice it is usually feasible only for small and relatively simple algorithms. For more sizeable and complicated algorithms verification is exceedingly difficult, even if it is only restricted to soundness. There are too many details to deal with, and as the complexity of the algorithm grows so does the complexity of the proof, to the point where the effort we invest in the proof might come to equal or exceed the effort that went into the algorithm. On the positive side, a general proof has a fixed, constant cost. Once the proof is completed and verified, we can confidently execute the algorithm as many times as we want with zero additional effort. By contrast, we will see that the model of certified computation we propose here has a run-time price: the algorithm has to do extra work to certify its results every time we use it. But this is usually easier to achieve than general verification.

1.1.3 The deductive approach

A drastically different alternative, and the one we will advocate here, is the relational, or deductive approach: use inference instead of computation, and do away with the algorithm f_R altogether. That is, instead of having an algorithm f_R that takes $x_1, \ldots, x_n$ and produces some value y, have instead an inference method that takes $x_1, \ldots, x_n$ and *proves a theorem of the form* $R(x_1, \ldots, x_n, y)$, on the basis of certain axioms about R. This conveys much more useful information than the algorithm f_R can give us. All f_R can do is output a value

y; we have no idea whether or not that value is related to the inputs $x_1, \ldots, x_n$ in the desired manner. By contrast, the deductive approach yields not just a value y, but also an assertion of correctness, namely, that y is related to $x_1, \ldots, x_n$ via R. If our inference method is *sound*, i.e., if it can only generate statements that follow logically from the axioms, then, modulo the truth of the axioms, the said assertion will hold: y will indeed be a correct answer. The difference between the two approaches, algorithmic and deductive, may be depicted graphically as follows:

	Input	Output
Computation (algorithm f_R):	$x_1, \ldots, x_n$	A value y
Deduction (method m_R):	$x_1, \ldots, x_n$	A theorem $R(x_1, \ldots, x_n, y)$

The idea of using deduction for computational purposes is not new. It is the cornerstone of the school of *relational logic programming*, which dates back at least to the inception of Prolog in the early 1970s. The term "relational" is meant to emphasize the shift of focus from functions (algorithms f_R) to relations (R itself). In this setting the program is a theory—usually first-order, though not necessarily. The programmer is concerned with a given problem domain, which consists of a certain collection of objects, operations on those objects, and relations amongst them. A logic vocabulary is introduced for the purpose of referring to these entities, and then a first-order theory is set up, as a collection of axioms and inference rules, that describes whatever aspects of the problem domain are of interest. It is of paramount importance that this theory be consistent. In particular, the problem domain (which is "the intended interpretation"), when viewed as an abstract relational structure, should be a *model* of the theory.

Computation in such a framework amounts to *deriving theorems*. In other words, we deduce logical consequences of our theory. The computational value of this scheme lies in constructive proofs of existentially quantified statements, since finding a y such that $R(x_1, \ldots, x_n, y)$ is reducible to having a constructive proof of the statement

$$(\exists\, y)\, R(x_1, \ldots, x_n, y). \tag{1.2}$$

Computation in this setting is described by the well-known slogan of Kowalski:

$$\textit{Computation} = \textit{Logic} + \textit{Control}.$$

The *Logic* part is our theory: a collection of axioms and inference rules allowing us to produce conclusions of the form $R(x_1, \ldots, x_n, x_{n+1})$. Once we specify this part, the set of theorems that can be derived becomes fixed. The *Control* part amounts to a theorem-proving strategy: *how* we plan to use the logic part in order to prove 1.2, once we are given some inputs $x_1, \ldots, x_n$. The chief requirement on the control part is soundness: if our control engine produces a proposition $R(x_1, \ldots, x_n, y)$, that proposition must be derivable from the axioms and the inference rules specified in the logic part.

In logic programming, however, the control part is fixed and immutable. It is baked into the underlying framework and cannot be extended. Users cannot specify and use arbitrary proof strategies of their own devising. In other words, they cannot determine *how* the proof effort will proceed, given the inputs. They can only specify the logic part, and they can do that only with axioms, not with arbitrary inference rules, since that would impact the control engine. That is a deliberate choice, made in the quest for *declarative* problem-solving, whose central tenet is that the users should only have to state the facts of the problem (the "what"), and let the system do the rest (take care of the "how"). The holy grail of the declarative approach is for the user to describe the relation R via axioms, formulate a query $R(x_1, \ldots, x_n, ?y)$, sit back, and eventually get an instantiated answer $?y \mapsto \cdots$, or an indication to the effect that $(\exists\, y)\, R(x_1, \ldots, x_n, y)$ does not follow from the axioms.

Certified computation with DPLs is quite different in that the user is given unrestricted freedom in structuring the control part. The framework does not offer any built-in proof mechanisms. Rather, it offers mechanisms for conveniently expressing arbitrary proofs and proof methods. This admittedly represents a shift from the declarative to the procedural, but it has several advantages: generality, adaptability, and extensibility. We will elaborate these in Section 1.4, but observe that the declarative idea is not lost: a powerful inference mechanism can be built and then offered as a library that people can use to solve problems declaratively. For instance, we can implement linear resolution as a proof strategy and then offer that as a module that people can use to do customary logic programming: simply write down declarative Horn clauses, add them to the assumption base, and let the resolution method do the proof search. This has been implemented in Athena. In addition, our framework allows for the explicit construction of certificates: the system does not only produce a theorem $R(x_1, \ldots, x_n, y)$, but also a *proof* of $R(x_1, \ldots, x_n, y)$. This means that we no longer have to trust the control part of the system (the part that performs the proof search): as long as we are given a proof, we do not care how the proof was obtained. We will discuss these points in more detail later on, but first let us look at an example.

1.2 A simple example

As a very simple example, consider the following problem: given any natural number n, find the parity of n.

We start by defining the natural numbers as a simple inductive structure:

```
(structure Nat
  zero
  (succ Nat))
```

We may now identify the natural numbers with the ground *terms* we can built from the constructors of this structure: `zero`, `(succ zero)`, `(succ (succ zero))`, and so on. Next we introduce two unary relation symbols, `Even` and `Odd`, which are intended to assert that a given natural number is even (or odd, respectively). In Athena, relation and function symbols are strongly typed ("sorted"), and may be polymorphic. Relation symbols are simply function symbols having `Boolean` as their range. We thus issue the following declarations:

```
(declare Even (-> (Nat) Boolean))

(declare Odd (-> (Nat) Boolean))
```

Intuitively, the first declation says that `Even` is a function that takes a natural number and produces either `true` or `false` (`Boolean` is a predefined structure that has only two elements, `true` and `false`); and likewise for `Odd`.

At this point we only have a domain of values, `Nat`, and two completely uninterpreted unary relations on them, `Even` and `Odd`. To make things interesting we need to specify the semantics of these two relations. We do this by postulating a small number of axioms. Our first axiom will simply assert that `zero` is even:

```
(define zero-axiom (Even zero))
```

Our next axiom specifies that if any number n is even, then its successor is odd:

```
(define odd-succ-axiom
  (forall ?n
    (if (Even ?n)
        (Odd (succ ?n)))))
```

Note that an object-level variable in Athena consists of a questionmark `?` followed by an identifier. Further, although variables are sorted, the user does not need to explicitly indicate their sorts; Athena figures that out automatically. In this case, Athena would infer that the sort of the variable `?n` is `Nat`.

Finally, we define one more axiom that allows us to conclude that non-zero numbers are `Even`:

```
(define even-succ-axiom
  (forall ?n
    (if (Odd ?n)
        (Even (succ ?n)))))
```

Our logic is done. All that is left now is to *assert* these axioms, i.e., to postulate them. This is done with the directive `assert`. An assertion is of the form

$$\texttt{(assert } P_1 \cdots P_n\texttt{)}$$

and its effect is to add the propositions $P_1, \ldots, P_n$ to the current assumption base. We thus write:

```
(assert zero-axiom odd-succ-axiom even-succ-axiom)
```

At this point we have our first-order theory. Let us now see how we can use this theory for computational purposes. Our task is to write a method (a "proof strategy") `infer-parity`, that will take any `Nat` object n as input and will derive a result of the form `(Even` n`)` or `(Odd` n`)`. In pseudo-code, our method might proceed as follows:

- If n is `zero`, invoke `zero-axiom` to conclude `(Even zero)`.

- Otherwise, if n is of the form `(succ` k`)`, apply `infer-parity` recursively to k. This will presumably result in a theorem of the form `(Even` k`)` or `(Odd` k`)`. Without loss of generality, suppose that the first is the case, i.e. that the recursive call has proven that k is even. Call that theorem T. Now, having proven `(Even` k`)`, we can easily prove that `(succ` k`)`—namely, n—is odd, via `odd-succ-axiom`, in two steps: first, instantiating `odd-succ-axiom` with k in place of `?`x gives the conditional `(if (Even` k`) (Odd (succ` k`)))`; then applying modus ponens to this conditional and theorem T yields the result `(Odd (succ` k`))`. Similar reasoning applies when the recursive call yields the theorem `(Odd` k`)` instead of `(Even` k`)`.

We can straightforwardly transcribe this into Athena code as follows:

```
(define (infer-parity n)
  (dmatch n
    (zero (!claim zero-axiom))
    ((succ k) (dlet ((T (!infer-parity k)))
                (dmatch T
                  ((Even _) (!mp (!uspec odd-succ-axiom k) T))
                  ((Odd _)  (!mp (!uspec even-succ-axiom k) T)))))))
```

Here `uspec` is Athena's primitive inference rule for universal quantification: `uspec` takes two arguments, a proposition P and a term t. If P is of the form `(forall ?`I Q`)` and is contained in the current assumption base; and if the term t has a sort that is consistent with the sort of the variable `?`I in Q; then the result of `uspec` is the proposition obtained from Q by replacing every free occurence of `?`I by t.[2] Otherwise `uspec` generates an error. The primitive method `mp` is Athena's modus ponens rule. It expects two propositions as arguments, which must be in the current assumption base and must be of the form `(if` P Q`)` and P, respectively. In that case, `mp` yields the conclusion Q; otherwise an error is generated. Finally, `dmatch` is a special form for "deductive matching", comprising a discriminant phrase M and a sequence of *pattern-deduction* pairs $(\pi_1\ D_1) \cdots (\pi_n\ D_n)$. First M is evaluated, producing a value V. That value is then successively matched against the patterns $\pi_1, \pi_2, \ldots$, until a match is found. When a π_i is found that matches V, deduction D_i is performed, under whatever bindings were produced as a result of matching V against π_i. The result of the entire `dmatch` is then the result of D_i. An error is generated if no match is found.

We can now apply the method `infer-parity` to an arbitrary number to deduce its parity, e.g.,

```
>(!infer-parity (succ (succ (succ zero))))

Theorem: (Odd (succ (succ (succ zero))))

>(!infer-parity (succ (succ zero)))

Theorem: (Even (succ (succ zero)))
```

[2] Taking care to perform alphabetic renaming in order to avoid any variable captures.

Despite its simplicity, the behavior of `infer-parity` exhibits a pattern that is typical of much more complicated methods. The run-time evaluation of such a method call typically consists of two stages:

1. *Backwards search* (*analysis*): During this stage we seek to decompose a given input into $k > 0$ simpler inputs such that if and when we have managed to prove appropriate results $T_1, \ldots, T_k$ for these simpler inputs, then we can prove an appropriate theorem T for the original input by applying some primitive method to the "lemmas" $T_1, \ldots, T_k$. The decomposition is typically performed by applying the method recursively to the smaller inputs.

2. *Forward proof* (*synthesis*): Once we have reached an input that cannot be simplified any further, we usually invoke some axiom that proves an appropriate statement for that input. Then the stack of recursive method calls piled up during the previous stage begins to unwind, and as we move back up the recursion tree we keep applying appropriate primitive methods to the results that were obtained recursively. This essentially builds a regular forward proof on the fly, whose final conclusion is the desired theorem about the original input. It is the "justification" stage of the certified computation, and where the payoff of deduction is realized: every result generated during this stage is a lemma that follows logically from the axioms and the primitive inference rules of our logic. This is guaranteed by the semantics of DPLs, which ensure that the assumption base is properly threaded throughout.

We can depict the flow of control in this two-stage process for a typical method call such as `(!infer-parity (succ (succ zero)))` with a diagram like the one shown in Figure 1.1. Evaluation begins at the upper left corner and proceeds counter-clockwise. The downward arrows on the left side correspond to recursive calls; they represent the analytic stage of input decomposition. The upward arrows on the right side correspond to primitive method calls; they represent the synthetic part of constructing the proof.

The semantics of Athena are such that if any method produces a proposition P in an assumption base β, then P can be proved in β by primitive methods only. This means that if the primitive methods are sound and β contains true propositions, then the result P will also be true—it will be a logical consequence of β. That, as we mentioned before, is the fundamental theorem of the $\lambda\mu$ calculus, and is the reason why every time Athena reports the result of a deduction, it confidently pronounces it a theorem. However, we do not have to take this at face value: we can ask Athena to explicitly produce this detailed proof that uses nothing but primitive methods and the axioms of the assumption base. This is done by issuing the command `expand-next-proof`. Then the next deduction that the user enters will be expanded into a long primitive proof, which will be assigned to the system variable `last-proof`:

```
>(expand-next-proof)

OK.
```

$$
\begin{tikzcd}
\texttt{(!infer-parity (succ (succ zero)))} \arrow[d] & \texttt{(Even (succ (succ zero)))} \\
\texttt{(!infer-parity (succ zero))} \arrow[d] & \texttt{(Odd (succ zero))} \arrow[u, "\texttt{uspec, mp}"'] \\
\texttt{(!infer-parity zero)} \arrow[r, "\texttt{zero-axiom}"'] & \texttt{(Even zero)} \arrow[u, "\texttt{uspec, mp}"']
\end{tikzcd}
$$

Figure 1.1: Control flow, shown counter-clockwise, for the method call `(!infer-parity (succ (succ zero)))`.

```
>(!infer-parity (succ (succ zero)))

Theorem: (Even (succ (succ zero)))

>(print last-proof)

(dbegin
  (!claim (Even zero))
  (!uspec (forall ?n (if (Even ?n) (Odd (succ ?n)))) zero)
  (!mp (if (Even zero) (Odd (succ zero)))
       (Even zero))
  (!uspec (forall ?n (if (Odd ?n) (Even (succ ?n)))) (succ zero))
  (!mp (if (Odd (succ zero)) (Even (succ (succ zero))))
       (Odd (succ zero))))

(): Unit
```

Sure enough, when we ask Athena to print the result of the expansion, we get a long proof that uses nothing but universal instantiation (`uspec`), modus ponens (`mp`), and the three axioms `zero-axiom`, `odd-succ-axiom`, and `even-succ-axiom`.[3] This is the *certificate* that is produced by applying the method `infer-parity` to the input `(succ (succ zero))`, and can be regarded as the justification for the result

$$\texttt{(Even (succ (succ zero)))}. \tag{1.3}$$

Observe that the above certificate is precisely what a human would adduce to convince someone that `(succ (succ zero))` is even. The reasoning is as follows: to begin with, `zero` is even. But if `zero` is even then `(succ zero)` is odd (this follows from the first universal instantiation

[3]Strictly speaking, the special form `dbegin` is also used in order to compose the intermediate inferences, ensuring that the assumption base is properly threaded from each inference to the next. As a construct for putting proofs together, it is the lowest common denominator offered by a DPL. It is analogous to the proof composition operator ; of $\mathcal{CND}$.

and application of modus ponens). And if `(succ zero)` is odd, then `(succ (succ zero))` is even (the second universal instantion and modus ponens). In more traditional notation:

1. `zero` is even.	By `zero-axiom`.
2. If `zero` is even then `(succ zero)` is odd.	Specialize `odd-succ-axiom`.
3. `(succ zero)` is odd.	2, 1, modus ponens.
4. If `(succ zero)` is odd then `(succ (succ zero))` is even.	Specialize `even-succ-axiom`.
5. `(succ (succ zero))` is even.	4, 3, modus ponens.

Now that we have the certificate, we can go ahead and check it, to reconfirm the result:

```
>(dbegin
   (!claim (Even zero))
   (!uspec (forall ?n (if (Even ?n) (Odd (succ ?n)))) zero)
   (!mp (if (Even zero) (Odd (succ zero)))
        (Even zero))
   (!uspec (forall ?n (if (Odd ?n) (Even (succ ?n)))) (succ zero))
   (!mp (if (Odd (succ zero)) (Even (succ (succ zero))))
        (Odd (succ zero))))

Theorem: (Even (succ (succ zero)))
```

Note that checking the certificate is considerably more efficient than discovering it. In essence, we have eliminated the effort on the left-hand side of Figure 1.1, which is half of the work that `infer-parity` does (the analytic half). Also note that Athena does not construct explicit certificates by default. Instead, a certificate is tacitly constructed and validated every time we evaluate a deduction. Consider, for instance, the evaluation of the method call `(!infer-parity (succ (succ zero)))`, as depicted in Figure 1.1. As the call stack unwinds, on the right-hand side of the picture, we are essentially building and simultaneously validating the relevant certificate on the fly. In that light, it is not unreasonable to view the call `(!infer-parity (succ (succ zero)))` itself as the certificate of the result 1.3. This view is encouraged by the design of the language, since a phrase such as `(!infer-parity (succ (succ zero)))` is, syntactically, a deduction, not an expression; and it is fully backed up by the semantics, which ensure that a method call can only produce a result that follows from primitive methods and the assumption base. But if we need to convince a skeptic, or if we are striving for a minimal trusted computing base, or if we just wish to satisfy our own curiosity, we have the option of requesting an explicit certificate, via `expand-next-proof`, and Athena will be happy to oblige.

The main practical drawback of `infer-parity`, and of certified computation in general, is the computational expense incurred by deduction. The difference from a conventional algorithm is intuitively easy to understand: a conventional algorithm will simply generate the result, whereas a certifying algorithm will both generate *and* justify it. The justification effort is the extra cost of certification. For instance, contrast the method `infer-parity` given above with a plain parity-computing algorithm:

```
(define (parity n)
```

```
(match n
  (zero 'Even)
  ((succ k) (match (parity k)
              ('Even 'Odd)
              ('Odd  'Even)))))
```

The recursive decomposition of a given input is the same here as it was in the case of `infer-parity`. However, the difference becomes clear during the synthetic part of putting the solution together, once we reach the base case `zero` and start moving back up the recursion tree. As the call stack of `parity` unwinds, all we have to do at each step is simply flip one bit, from `'Even` to `'Odd` and vice versa. By contrast, as the call stack of `infer-parity` unwinds, we are actually constructing a proof: at each step, we have to apply inference rules to certain premises and generate the appropriate conclusions, which will serve as premises at some future step.

Note, however, that the total expense of justification is proportional to the complexity of the conventional algorithm. Justification only adds a comparable amount of extra work at each step of the process—in this case a constant amount. It does not blow up the overall asymptotic complexity by any non-trivial (non-constant) factor, e.g., from $\log n$ to n, or from n to n^2, etc. According to our experience, that appears to be the case in most situations, not just in the simple parity example. But that is an empirical observation; we have not addressed the question formally. Still, whether or not the overhead is acceptable depends on the situation at hand. For performance-conscious applications it might not be an option. But for correctness-critical applications, or in situations where we need to execute code supplied by an untrusted source, it will be a reasonable price to pay for the gained assurance.

In any event it becomes clear that a desired property of any platform for certified computation is *partiality*: We should be able to apply this methodology to selected *parts* of a software system rather to the whole thing. Certified computation should not be all-or-nothing. The motivation for partiality is efficiency: in many cases it will be impractical to certify the result of the entire computation. However, we might still profitably seek some extra assurance by applying logical certification to certain safety-critical modules. For example, compiler writers are not likely to certify lexical analyzers and parsers. These parts are so well understood and so routinely automated by highly reliable tools that it would be counter-productive to expend too much effort on them. However, we might still want to certify the optimizing part of the compiler's back end, or a type inference algorithm, or some other module whose correctness is crucial. DPLs are ideal for such partial certification thanks to their natural integration of deduction and computation. All we have to do is write the module we wish to certify as a method, rather than a function. Then every call to it becomes a method call, which represents a deduction, rather than a function call (which represents an arbitrary computation). Method calls can appear anywhere within a computation; the semantics of the $\lambda\mu$-calculus ensure that the results are soundly derived, and seamlessly integrated into any further computation, just as if they had been generated by conventional procedure calls.

1.3 Another example: unification

As a more involved example, in this section we will use certified computation to implement first-order unification. A conventional unification algorithm takes as input two terms s and t and produces a most general unifier ("mgu") θ for them, if one exists; otherwise the algorithm fails. By contrast, our unification prover will take s and t as inputs and will not simply produce θ, but will in fact *prove* that θ is an idempotent mgu for s and t. In other words, the output of our prover will be a theorem of the form $imgu(s, t, \theta)$, asserting that θ is an idempotent mgu of s and t.

The first step is to formulate a logic—a collection of axioms and inference rules—that allows us to derive judgments of this form. The rules should be sound, i.e., they should not allow us to "prove" that some θ is a mgu for s and t when in fact it is not. Then, based on this logic, we can start to implement a unification procedure as a DPL method.

For the remainder of this section, fix a set $\mathcal{F}$ of function symbols, each with a unique non-negative arity, and a disjoint set $\mathcal{V}$ of variables. In what follows, by "term" we will mean a first-order Herbrand term over $\mathcal{F}$ and $\mathcal{V}$: either a variable or an "application" $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$ has arity n and each t_i is a term. By "equation" we will mean an ordered pair of terms $\langle s, t\rangle$, which will be more suggestively written as $s \approx t$; and by "substitution" we will mean a function from $\mathcal{V}$ to the set of terms over $\mathcal{F}$ and $\mathcal{V}$ that is the identity almost everywhere. We use the letters x, y, and z as typical variables; f, g, and h as function symbols; s and t for terms; and θ, σ, and τ for substitutions. We write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for the substitution that maps each x_i to t_i (we assume that the variables $x_1, \ldots, x_n$ are distinct), and every other variable to itself; and we write $\overline{\theta}$ for the unique homomorphic extension of a substitution θ.

1.3.1 A conventional algorithm

Martelli and Montinari in 1982 [7] introduced a particularly elegant 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 [5] in the 1930s. 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 here we will mean a list of the form

$$E = \langle s_1 \approx t_1, \ldots, s_n \approx t_n\rangle. \tag{1.4}$$

We write E_1, E_2 for the list obtained by concatenating E_1 and E_2. For convenience, we will sometimes treat a single equation as an one-element list, e.g. writing $E, s \approx t$ instead of $E, \langle s \approx t\rangle$. Finally, for any substitution θ and system E of the form 1.4, we write $\overline{\theta}(E)$ for the system

$$\langle\overline{\theta}(s_1) \approx \overline{\theta}(t_1), \ldots, \overline{\theta}(s_n) \approx \overline{\theta}(t_n)\rangle.$$

Recall that a system of equations E of the form 1.4 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 θ is more general than 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 composition with a renaming, 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 $\langle s \approx t\rangle$ is unifiable.

A system E is said to be *in solved form* iff the set of equations that occur in E is of the form $\{x_1 \approx t_1, \ldots, x_n \approx t_n\}$, where the variables $x_1, \ldots, x_n$ are distinct and x_i does not occur in t_j for any $i, j \in \{1, \ldots, n\}$.[4] It is straightforward to show that a system E in solved form determines a unique substitution

$$\theta_E = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

that is an idempotent most general unifier of E.

The HMM algorithm attempts to transform a given set of equations into solved form by repeated applications of the following rules:

- *Simplification*: $E_1, t \approx t, E_2 \Longrightarrow E_1, E_2$;
- *Decomposition*: $E_1, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n), E_2 \Longrightarrow E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2$;
- *Transposition*: $E_1, t \approx x, E_2 \Longrightarrow E_1, x \approx t, E_2$—provided that t is not a variable;
- *Application*: $E_1, x \approx t, E_2 \Longrightarrow \overline{\{x \mapsto t\}}(E_1), x \approx t, \overline{\{x \mapsto t\}}(E_2)$—provided that x occurs in E_1, E_2 but not in t.

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_1, E_2 in the last rule (the second qualification of that rule also ensures that the process does not proceed in the presence of an equation $x \approx t$ where x occurs in 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 point will be made clear when we come to build our theorem prover, at which point 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 $E_1 = \langle s \approx t\rangle$ and keep applying rules (nondeterministically), 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 not difficult to

[4] Note that this definition allows E to have multiple occurences of an equation.

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 $E_k = \langle x_1 \approx t_1, \ldots, x_n \approx t_n \rangle$, 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 $f(a, y, b, h(x))$:

$$
\begin{array}{lll}
1. & \langle f(x, g(z), b, z) \approx f(a, y, b, h(x)) \rangle & \Longrightarrow \quad \textit{Decompose} \\
2. & \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(x) \rangle & \Longrightarrow \quad \textit{Apply } x \approx a \\
3. & \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(a) \rangle & \Longrightarrow \quad \textit{Transpose} \\
4. & \langle x \approx a, y \approx g(z), b \approx b, z \approx h(a) \rangle & \Longrightarrow \quad \textit{Simplify} \\
5. & \langle x \approx a, y \approx g(z), z \approx h(a) \rangle & \Longrightarrow \quad \textit{Apply } z \approx h(a) \\
6. & \langle x \approx a, y \approx g(h(a)), z \approx h(a) \rangle &
\end{array}
$$

The system $\langle x \approx a, y \approx g(h(a)), z \approx h(a) \rangle$ 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.

1.3.2 A logic for proving unification judgments

We will now set up 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 $\langle s \approx t \rangle$ 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.

$$\frac{}{\vdash_U \langle x_1 \approx t_1, \ldots, x_n \approx t_n \rangle : \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}} \; [\textit{Solved-Form}]$$

provided $\langle x_1 \approx t_1, \ldots, x_n \approx t_n \rangle$ is in solved form

$$\frac{\vdash_U E_1, E_2 : \theta}{\vdash_U E_1, t \approx t, E_2 : \theta} \; [\textit{Reflexivity}]$$

$$\frac{\vdash_U E_1, s \approx t, E_2 : \theta}{\vdash_U E_1, t \approx s, E_2 : \theta} \; [\textit{Symmetry}]$$

$$\frac{\vdash_U E_1, s_1 \approx t_1, \ldots, s_n \approx t_n, E_2 : \theta}{\vdash_U E_1, f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n), E_2 : \theta} \; [\textit{Congruence}]$$

$$\frac{\vdash_U E_1, x \approx t, E_2 : \theta}{\vdash_U E_1', x \approx t, E_2' : \theta} \; [\textit{Abstraction}]$$

provided $\overline{\{x \mapsto t\}}(E_1', E_2') = E_1, E_2$.

Figure 1.2: A logic for deducing idempotent most general unifiers.

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, to write a method that takes two terms s and t and—provided that s and t are unifiable—returns a theorem of the form $\langle s \approx t \rangle : \theta$.

The logic comprises one axiom and four unary rules, shown in Figure 1.2 The axiom [*Solved-Form*] asserts that every system of equations $E = \langle x_1 \approx t_1, \ldots, x_n \approx t_n \rangle$ in solved form is unifiable, and that, in particular, $\theta_E = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent mgu of E. The rules [*Reflexivity*], [*Symmetry*], and [*Congruence*] are self-explanatory, and their soundness should be clear (it is straightforward to prove formally that all five rules are sound [1]). 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.

Finally, consider the rule [*Abstraction*]. The key here is the proviso

$$\overline{\{x \mapsto t\}}(E'_1, E'_2) = E_1, E_2.$$

This means that the equations in E'_1, E'_2 are abstractions of the equations in E_1, E_2 obtainable from the latter by replacing certain occurences of t by x. Alternatively, the equations in E_1, E_2 are *instances* of the equations in E'_1, E'_2, 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'_1, E'_2 are *more general* than those of E_1, E_2; 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.	$\vdash_U \langle x \approx a, y \approx g(h(a)), z \approx h(a)\rangle : \theta$	[*Solved-Form*]
2.	$\vdash_U \langle x \approx a, y \approx g(z), z \approx h(a)\rangle : \theta$	1, [*Abstraction*] on $z \approx h(a)$
3.	$\vdash_U \langle x \approx a, y \approx g(z), b \approx b, z \approx h(a)\rangle : \theta$	2, [*Reflexivity*]
4.	$\vdash_U \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\rangle : \theta$	3, [*Symmetry*]
5.	$\vdash_U \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\rangle : \theta$	4, [*Abstraction*] on $x \approx a$
6.	$\vdash_U \langle f(x, g(z), b, z) \approx f(a, y, b, h(x))\rangle : \theta$	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.

1.3.3 Implementation

In this section we implement our unification logic in Athena, giving complete working code that the readers can use for experimentation. First we must decide on a representation for terms. We choose the following simple representation:

```
(structure Term
  (Var Ide)
  (App Ide (List-Of Term)))
```

where `List-Of` is the following polymorphic structure:

```
(structure (List-Of T)
  Nil
  (Cons T (List-Of T)))
```

and `Ide` is a pre-defined domain of "identifiers". This domain is used whenever we wish to represent a set of object-level variables. Every string consisting of a single quote followed by an Athena name is a constant of sort `Ide`: `'X`, `'a-long-identifier`, `'foo`, etc. Since these are constant Athena symbols, we can reason about them, e.g., we can quantify over them:

```
>(exists ?I (= ?I 'foo))

Proposition: (exists ?I:Ide (= ?I 'foo))
```

Thus a variable such as x is represented as `(Var 'x)`; a constant a is represented by the term `(App 'a Nil)`; and an application such as $f(x, a)$ by

```
(App 'f (Cons (Var 'x) (Cons (App 'a Nil) Nil))).
```

The following simple structure models an equation between two terms:

```
(structure Equation
  (== Term Term))
```

Thus the term `(== (Var 'x) (App 'a Nil))` represents the equation $x \approx a$. The function `equate` given below will come handy later: it takes two lists of terms $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$ and "zips" them into a list of equations $s_1 \approx t_1, \ldots, s_n \approx t_n$:

```
(define (equate terms1 terms2)
  (match [terms1 terms2]
    ([Nil Nil] Nil)
    ([(Cons s rest1) (Cons t rest2)] (Cons (== s t) (equate rest1 rest2)))))
```

We will also need the following five classic list functions:

```
(define (map f l)
  (match l
    (Nil Nil)
    ((Cons x rest) (Cons (f x) (map f rest)))))

(define (append l1 l2)
  (match l1
    (Nil l2)
    ((Cons x rest) (Cons x (append rest l2)))))

(define (for-each l f)
  (match l
    (Nil true)
    ((Cons x rest) (& (f x) (for-each rest f)))))
```

```
(define (for-some l f)
  (match l
    (Nil false)
    ((Cons x rest) (|| (f x) (for-some rest f)))))

(define (reverse l)
  (letrec ((rev (function (l1 l2)
                  (match l1
                    (Nil l2)
                    ((Cons x rest) (rev rest (Cons x l2)))))))
    (rev l Nil)))
```

Note that `&` and `||` are special Athena forms for short-circuit evaluation of "and" and "or".[5] We also define a negation function that maps `true` to `false` and vice versa:

```
(define (~ b)
  (match b
    (true false)
    (false true)))
```

A substitution is modelled as a list of ordered pairs, each of which consists of an identifier and a term:

```
(structure (Pair-Of S T)
  (Pair S T))

(define Substitution (List-Of (Pair-Of Ide Term)))
```

For instance, the substitution $\{x \mapsto a, y \mapsto f(z)\}$ is represented by

```
(Cons (Pair 'x (App 'a Nil))
      (Cons (Pair 'y (App 'f (Cons (Var 'z) Nil)))
            Nil))
```

By convention, substitution-representing lists grow on the left, so to find the value of a substitution `theta` for a given variable, say `(Var 'x)`, we scan the list `theta` from left to right until we find a pair of the form `(Pair 'x` t`)`. If we find such a pair, then t is the desired value; otherwise the substitution is undefined on the given variable. The function `apply-sub-to-var` implements this:

```
(define (apply-sub-to-var sub x)
  (match sub
    (Nil (Var x))
    ((Cons (Pair (val-of x) t) _) t)
    ((Cons _ rest-sub) (apply-sub-to-var rest-sub x))))
```

[5] Both of these forms perform computation on the two-element set {`true`,`false`}, and should not be confused with the propositional constructors `and` and `or`.

The higher-order function `lift` below defines the homomorphic extension $\overline{\theta}$ of a given substitution θ. For convenience, we define $\overline{\theta}$ so that it can be applied not just to a single term, but to an equation $s \approx t$ as well, producing the equation $\overline{\theta}(s) \approx \overline{\theta}(t)$:

```
(define (lift sub)
  (function (t)
    (match t
      ((Var x) (apply-sub-to-var sub x))
      ((App f args) (App f (map (lift sub) args)))
      ((== t1 t2) (== ((lift sub) t1) ((lift sub) t2))))))
```

Next we write a function `occurs` that takes a variable x and a term t and returns `true` if x occurs in t and `false` otherwise. Again for convenience, we write `occurs` so that it can also take an equation $t_1 \approx t_2$ instead of a single term, in which case it will return `true` if x occurs in t_1 or in t_2, and `false` if it occurs in neither:

```
(define (occurs x t)
  (match t
    ((Var (val-of x)) true)
    ((Var _) false)
    ((App _ terms) (for-some terms (function (s) (occurs x s))))
    ((== t1 t2) (|| (occurs x t1) (occurs x t2)))))
```

Systems of equations are modelled by lists:

```
(define System (List-Of Equation))
```

The following function returns `true` or `false` depending on whether or not the given system is in solved form:

```
(define (solved? E)
  (for-each E
    (function (eq)
      (match eq
        ((== (Var x) s) (for-each E
                          (function (eq)
                            (match eq
                              ((== (Var (val-of x)) t) (& (equal? s t)
                                                          (~ (occurs x t))))
                              ((== _ t) (~ (occurs x t)))))))
        (_ false)))))
```

Next we introduce a relation symbol `imgu` that is predicated of an equation system and a substitution:

```
(declare imgu (-> (System Substitution) Boolean))
```

A proposition `(imgu` E θ`)` is intended to express the judgment $\vdash_U E : \theta$ of our unification calculus. It asserts that θ is an idempotent mgu of E.

We are now in a position to introduce primitive Athena methods modelling the inference rules of that calculus. Unlike a defined method, whose body has to be a deduction, a primitive method has an expression as a body, which means that it can perform any computation it wishes on its arguments, as long as it eventually produces a proposition. Once defined, primitive methods can be used just as defined methods. Primitive methods are thus a very powerful mechanism, allowing us to formulate inference rules with arbitrarily complicated behaviors. Just like axioms, they should not be abused. We should always be able to convince ourselves (and others!) that a primitive method application will never produce a proposition that is not logically entailed by the assumption base in which the application takes place.[6]

We begin with a primitive method `solved-form` that models the corresponding inference rule of $\vdash_U$. This method takes a system E, and if E is in solved form $\langle x_1 \approx t_1, \ldots, x_n \approx t_n \rangle$, then it produces a proposition asserting that the substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is an idempotent mgu of E:

```
(primitive-method (solved-form E)
  (check ((solved? E) (imgu E (make-sub E)))))
```

Here `make-sub` is a function that takes a system E in solved form and produces the unique substitution θ_E determined by it:

```
(define (make-sub E)
  (match E
    (Nil Nil)
    ((Cons (== (Var x) t) rest) (Cons (Pair x t) (make-sub rest)))))
```

The rest of the primitive methods are straightforward; they are listed in Figure 1.3. They all use the top-level Athena function `holds?`, which takes a proposition P and returns `true` if P is in the current assumption base and `false` otherwise. Note how closely these definitions capture the inference rules of Figure 1.2. Each method:

1. takes as parameters all the pieces of data that appear in the corresponding inference rule (e.g. `sym` takes E_1, E_2, s, t, and the substitution θ, which are precisely the "free variables" of rule [*Symmetry*] in Figure 1.2);

2. checks to make sure that the appropriate premises (if any) are in the assumption base, and that all relevant side conditions (if any) are satisfied;

3. and finally it produces its respective conclusion.

Also note that all of these primitive methods are non-recursive. They are not entirely trivial, meaning that they do perform some iterative computation, mainly via `append`, or in the case of `solved-form` via the helper function `solved?`, whose running time grows quadratically with the size of the input system. But no primitive method is defined in terms of itself, and if we trust `append`, `holds?`, `solved?`, `equate`, `map`, and `lift`, all of which are fairly innocuous,

[6] The slogan to remember: a proof is only as good as its axioms and primitive methods.

```
(primitive-method (reflex E1 E2 t sub)
  (check ((holds? (imgu (append E1 E2) sub))
                  (imgu (append E1 (Cons (== t t) E2)) sub))))

(primitive-method (sym E1 E2 s t sub)
  (check ((holds? (imgu (append E1 (Cons (== s t) E2)) sub))
                  (imgu (append E1 (Cons (== t s) E2)) sub))))

(primitive-method (cong E1 E2 f args1 args2 sub)
  (check ((holds? (imgu (append E1 (append (equate args1 args2) E2)) sub))
                  (imgu (append E1 (Cons (== (App f args1) (App f args2))
                                         E2)) sub))))

(primitive-method (abstract E1 E2 x t sub)
  (let ((sub' (Cons (Pair x t) Nil))
        (eq (== (Var x) t)))
    (check ((holds? (imgu (append (map (lift sub') E1)
                                  (Cons eq (map (lift sub') E2))) sub))
                    (imgu (append E1 (Cons eq E2)) sub)))))
```

Figure 1.3: The Athena representation of the inference rules of $\mathcal{U}$.

then we should be quite confident in these methods. This is in adherence to the principle of minimizing our trusted computing base. Because primitive methods are part of our trusted base, they should not be computationally expensive. The more involved a primitive method is, the more difficult it becomes to understand and use it correctly, i.e., the more difficult it becomes to trust.

Finally, the method `unify` is given in Figure 1.4. The auxiliary function `find-candidate` does much of the work. It takes a system E and decomposes it into three parts, a prefix E_1, an equation $s \approx t$, and a suffix E_2, such that $E_1, s \approx t, E_2$ matches the left-hand side of one of the four HMM transformation rules given in page 14. These three values are bundled together in an Athena list and returned as the result of `find-candidate`. If no such decomposition of the input system E exists, the empty Athena list `[]` is returned.

Let us see how this method handles our earlier example of the system

$$\langle f(x, g(z), b, z) \approx f(a, y, b, h(x)) \rangle :$$

```
>(define s
   (App 'f (Cons (Var 'x)
                 (Cons (App 'g (Cons (Var 'z) Nil))
                       (Cons (App 'b Nil)
                             (Cons (Var 'z) Nil))))))

Term s defined.

>(define t
    (App 'f (Cons (App 'a Nil)
```

```
(define (find-candidate E)
  (letrec ((search
             (function (remaining-list front)
               (match remaining-list
                 (Nil [])
                 ((Cons (bind eq (== t t)) rest)
                    [(reverse front) eq rest])
                 ((Cons (bind eq (== (App f args) (Var x))) rest)
                    [(reverse front) eq rest])
                 ((Cons (bind eq (== (App f args1) (App f args2))) rest)
                    [(reverse front) eq rest])
                 ((Cons (bind eq (== (Var x) t)) rest)
                    (check
                      ((& (for-some (append front rest)
                                    (function (e) (occurs x e)))
                          (~ (occurs x t)))
                        [(reverse front) eq rest])
                      (else (search rest (Cons eq front)))))))))
    (search E Nil)))

(define (unify E)
  (dmatch (find-candidate E)
    ([] (dcheck ((solved? E) (!solved-form E))))
    ([E1 (== t t) E2]
        (dmatch (!unify (append E1 E2))
          ((imgu _ sub) (!reflex E1 E2 t sub))))
    ([E1 (== (App f args) (Var x)) E2]
        (dmatch (!unify (append E1 (Cons (== (Var x) (App f args)) E2)))
          ((imgu _ sub) (!sym E1 E2 (Var x) (App f args) sub))))
    ([E1 (== (App f args1) (App f args2)) E2]
        (dmatch (!unify (append E1 (append (equate args1 args2) E2)))
          ((imgu _ sub) (!cong E1 E2 f args1 args2 sub))))
    ([E1 (== (Var x) t) E2]
        (dmatch (!unify (append (map (lift (Cons (Pair x t) Nil)) E1)
                                (Cons (== (Var x) t)
                                      (map (lift (Cons (Pair x t) Nil)) E2))))
          ((imgu _ sub) (!abstract E1 E2 x t sub))))))
```

Figure 1.4: The definitions of `find-candidate` and `unify`.

```
              (Cons (Var 'y)
                    (Cons (App 'b Nil)
                          (Cons (App 'h (Cons (Var 'x) Nil))))))))

Term t defined.

>(!unify (Cons (== s t) Nil))

Theorem:
(imgu (Cons (== (App 'f (Cons (Var 'x)
                              (Cons (App 'g
                                         (Cons (Var 'z) Nil))
```

$$\begin{tikzcd}
\texttt{!unify}\ \langle f(x,g(z),b,z) \approx f(a,y,b,h(x))\rangle \arrow[d, "\text{Decompose}"] \arrow[dr, phantom, "\xleftrightarrow{\text{Inverses}}"] & \langle f(x,g(z),b,z) \approx f(a,y,b,h(x))\rangle : \theta \\
\texttt{!unify}\ \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\rangle \arrow[d, "\text{Apply}"] & \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(x)\rangle : \theta \arrow[u, "\textsf{!cong}"'] \\
\texttt{!unify}\ \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\rangle \arrow[d, "\text{Apply}"] & \langle x \approx a, g(z) \approx y, b \approx b, z \approx h(a)\rangle : \theta \arrow[u, "\textsf{!abstract}"'] \\
\texttt{!unify}\ \langle x \approx a, g(h(a)) \approx y, b \approx b, z \approx h(a)\rangle \arrow[d, "\text{Transpose}"] & \langle x \approx a, y \approx g(z), b \approx b, z \approx h(a)\rangle : \theta \arrow[u, "\textsf{!sym}"'] \\
\texttt{!unify}\ \langle x \approx a, y \approx g(h(a)), b \approx b, z \approx h(a)\rangle \arrow[d, "\text{Simplify}"] & \langle x \approx a, y \approx g(z), z \approx h(a)\rangle : \theta \arrow[u, "\textsf{!reflex}"'] \\
\texttt{!unify}\ \langle x \approx a, y \approx g(h(a)), z \approx h(a)\rangle \arrow[r, "\textsf{!solved}"'] & \langle x \approx a, y \approx g(h(a)), z \approx h(a)\rangle : \theta \arrow[u, "\textsf{!abstract}"']
\end{tikzcd}$$

Figure 1.5: Control flow, shown counter-clockwise, for the method application **!unify** $\langle f(x,g(z),b,z) \approx f(a,y,b,h(x))\rangle$, where $\theta = \{x \mapsto a, y \mapsto g(h(a)), z \mapsto h(a)\}$.

```
                              (Cons (App 'b Nil)
                                    (Cons (Var 'z) Nil)))))
          (App 'f (Cons (Var 'y)
                        (Cons (App 'b Nil)
                              (Cons (App 'h (Cons (Var 'x) Nil))
                                    Nil)))))
      Nil)
(Cons (Pair 'x (App 'a Nil))
      (Cons (Pair 'y
                  (App 'g (Cons (App 'h (Cons (App 'a Nil) Nil))
                                Nil)))
            (Cons (Pair 'z (App 'h (Cons (App 'a Nil) Nil)))
                  Nil))))
```

Figure 1.5 depicts the run-time flow of control when `unify` is applied to this system. Observe the structural similarities with the diagram of Figure 1.1. In essence, every primitive method call validates the work of the corresponding recursive call. By the time the entire certificate has been constructed and checked, we have validated the original problem decomposition and proof search.

The important point in this case is that we do *not* have to trust `find-candidate` or unify, which are by far the two most complicated parts of the system. We only need to trust our five primitive methods. This becomes evident when we ask Athena to produce the relevant certificates. For instance, if we ask Athena to produce the certificate for the method call

```
(!unify (Cons (== s t) Nil))
```

we will obtain the exact same proof that was given in page 17, which only uses the primitive inference rules of our logic.

1.4 Comparison with other approaches

As we mentioned earlier, the idea of using deduction for computational purposes has been around for a long time. There are several methodologies predating DPLs that can be used for certified computation. In this section we will compare DPLs to logic programming languages and to theorem proving systems of the HOL variety.

Comparison with logic programming

The notion of "programming with logic" was a seminal idea, and its introduction and subsequent popularization by Prolog was of great importance in the history of computing. Although logic programming languages can be viewed as platforms for certified computation, they have little in common with DPLs. DPLs are languages for writing proofs and proof strategies. By contrast, in logic programming users do not write proofs; they only write assertions. The inference mechanism that is used for deducing the consequences of those assertions is fixed and sequestered from the user: linear resolution in the case of Prolog, some higher-order extension thereof in the case of higher-order logic programming languages [9, 2], and so on. This rigidity can be unduly constraining. It locks the user into formulating every problem in terms of the same representation (Horn clauses, or higher-order hereditary Harrop clauses [10], etc.) and the same inference method, even when those are not the proper tools to use. For instance, how does one go about proving De Morgan's laws in Prolog? How does one derive $\neg(\exists\, x)\, \neg P(x)$ from the assumption $(\forall\, x)\, P(x)$? Moreover, how does one write a schema that does this for any given x and P? How about higher-order equational rewriting or semantic tableaux? Although in principle more or less everything could be simulated in Prolog, for many purposes such a simulation would be formidably cumbersome.

A related problem is lack of extensibility. Users have no way of extending the underlying inference mechanism so as to allow the system to prove more facts or different types of facts.

The heart of the issue is how much control the user should have over proof construction. In logic programming the proof-search algorithm is fixed, and users are discouraged from tampering with it (e.g., by using impure control operators or clever clause reorderings). Indeed, strong logic programming advocates maintain that the user should have no control at all over proof construction. The user should simply enter a set of assertions, sit back, and let the system deduce the desired consequences. Advocates of weak logic programming allow the user some control (e.g., witness the prune operators and the commit mechanism of Gödel [6]), but the differences are not too great. DPLs, on the other hand, give the user complete and unrestricted control over the proof search, without giving up soundness.

In fact the inference engines of logic programming languages are so completely insulated from the user that not only they cannot be modified, but they cannot even be viewed. Say that a Prolog programmer enters a list of Horn clauses, evaluates a ground query, and magically gets a "yes" answer. Where is the proof? What is the justification for the "yes"? We can't tell. Knowing the theory of logic programming and trusting the implementation, we believe that the query follows logically from our clauses, and we might even have an idea as to why that is, but the system will not offer any concrete evidence. By contrast, in a DPL the inference engine is written by the user (with some minimal but crucial support from the language) and thus the user knows exactly how a fact was proved—because he was the one who proved it. The application of a user-written method *is* the proof of the application's result.

But we can go even further: DPL implementations allow the user to request and obtain fully explicit certificates, as we illustrated in the examples, by expanding the evaluation of any given deduction into an elementary sequence of primitive methods. For instance, we have written a Prolog interpreter as an Athena method. That interpreter not only verifies a given query, but can also produce a certificate to further back up the query's validity. That certificate is essentially a long natural-deduction derivation of the query that uses nothing but modus ponens and universal instantiation. All the search (unification, etc.) has been thrown away. The produced certificate is the *justification* of the result. We can examine it independently to convince ourselves that the result holds, or to gain some insight as to why it does.

This entails a drastic reduction in the size of our trusted base. Once we have obtained a certificate, we no longer have to trust the entire DPL implementation. Mechanisms such as pattern matching, recursive methods, conditional branching, etc., could be defectively implemented and yet the defects will be immaterial because these mechanisms are simply not used in the certificate.[7] The certificate only uses primitive methods, which are usually eminently simple. As long as we trust our primitive methods and our axioms, we can trust the entire result. By contrast, in logic programming we have to trust the entire implementation.

In addition, the exposure of the logic and its complete separation from the control is a modularity boon: many different control engines can be used with—plugged into—a single logic. This facilitates the interaction of a code consumer with arbitrary code producers. A code consumer interested in unification, for example, can publicize the five primitive methods of Section 1.3.3 as a formal theory and then solicit unification methods from arbitrary code

[7] This is similar, say, to a Java program that is written entirely in a very small and simple subset of Java, e.g., a sequence of binary additions and multiplications. To execute that program properly, we don't need to trust that our Java interpreter correctly implements inheritance or multithreading. All we need to trust is that it gets additions, multiplications, and assignment right. In fact since this Java subset is so small and simple, we can write our own interpreter for it and use that instead. This is analogous to full-blown DPLs and the certificates produced by them.

producers. An aspiring code producer can supply an arbitrary method M (control engine), which the consumer can use locally in an environment that contained the said methods and in the empty assumption base. If and when M produces a result, the consumer can safely accept and use that result, knowing that it has been certified. The generated certificate formally proves that the result respects—follows from—the consumer's policy. But if M is buggy or malicious then its evaluation will eventually generate a logical error and hence its results will never be used, simply because they will never be generated in the first place.

This enables completely open software systems. Arbitrary code producers can come along and offer smarter or more efficient methods. One might imagine, for example, an extensible compiler that would allow arbitrary users to provide custom-made dataflow analyzers, expressed as methods.[8] The logic would be posted by the compiler itself, and would consist of axioms and primitive methods for deriving statements such as "variable x is live at this point", or "this definition of y does not reach that statement", and so on. Different methods for proving data flow theorems could then be supplied by arbitrary sources and the compiler could run them confidently, knowing that their results would be sound. The capacity of such an open compiler to reason about dataflow would be arbitrarily sophisticated—unlike current compilers, which are necessarily limited on account of having a fixed set of dataflow algorithms built into them. Similar ideas apply to other compiler phases, such as traditional code optimizations. This theme has been pursued by Rinard et. al. in the "credible compilation" project [11], where an optimizer—say, for constant propagation—does not only produce a new program but also proves a bisimulation theorem relating the transformed program to the original.

Comparison with LCF systems

Theorem-proving systems of the LCF family [3], such as HOL [4], could also be used for certified computation. HOL, in particular, is a programming language (ML) augmented with an abstract data type `theorem` and soundness-preserving ways of producing values of that type. Manipulating theorems soundly was in fact the primary motivation behind Milner's pioneering work on secure type systems, which eventually led to the automatic type inference algorithm used in ML.

There is a broad sense in which systems of this kind are similar to DPLs: in both settings the user starts out with a piece of text that is regarded as a proof (or a proof-construction recipe), and evaluates that text in accordance with the formal semantics of the language in order to obtain a valid judgment. There are several significant differences, however.

Perhaps the most fundamental difference is that DPLs incorporate the abstraction of assumption bases into the underlying semantics of the language, and this results in an altogether different model of computation. If one were to give a formal denotational semantics for HOL, the top meaning function would have a signature of the form

$$M : \mathit{Phrase} \to \mathit{Env} \to \mathit{Store} \to \mathit{Cont} \to \mathit{Val}. \tag{1.5}$$

[8]This idea is due to Olin Shivers.

In other words, to obtain the value (meaning) $M[E]$ of a given HOL phrase E, we need an environment, a store, and a continuation. This semantic model is similar for many other conventional programming languages. The denotational semantics of Scheme, for example, have the exact same signature, 1.5, as HOL. This reflects the fact that HOL is, in an essential sense, a programming language.

By contrast, if we were to write a formal denotational semantics for a DPL such as Athena, the signature of the meaning function would be different:

$$M : \mathit{Phrase} \rightarrow \mathit{Env} \rightarrow \mathit{ABase} \rightarrow \mathit{Store} \rightarrow \mathit{Cont} \rightarrow \mathit{Val}. \tag{1.6}$$

What is included here but missing in the case of HOL or Scheme is the additional semantic parameter *ABase*—the assumption base. This is a set of propositions, where the exact specification of what counts as a proposition might vary from DPL to DPL. Assumption bases are *the* central abstaction in informal mathematical proofs, and weaving that absraction into the fundamental semantics of the language enables us to capture important patterns of informal reasoning (such as inference composition or assumption introduction and discharge) by introducing correspoding syntactic constructs and formally specifying their behavior in terms of how they manipulate the assumption base.

In fact DPLs separate the syntax of deductions from the syntax of computations. Proofs and computations are written in *different languages*, and have *different semantics*, even though the two can be seamlessly intertwined. Proofs are semantically constrained to return propositions. The proposition produced by evaluating a proof is naturally viewed as its conclusion. Computations, on the other hand, can return any type of value, including propositions, but no soundness guarantee is made about them. A soundness guarantee is only made for deductions, and always takes the following form: if a deduction D produces a proposition P in the context of an assumption base β, then P is a logical consequence of β. That is the fundamental theorem of the $\lambda\mu$ calculus. Observe that we would not even be able to state this theorem if it were not for these three facts:

1. computations and deductions are syntactically distinct;
2. deductions always return propositions; and
3. evaluation always takes place with respect to an assumption base.

None of these hold in the LCF world. Lacking assumption bases, HOL uses sequents for assumption management. Sequents have several drawbacks for writing proofs [1]. Moreover, because in HOL everything is a computation (a program), a static type system is needed to distinguish those values that represent theorems. That is not necessary in DPLs, owing to the syntactic separation of computations and proofs. (However, a DPL with a strong static type system is certainly possible.) There are other technical differences. In a DPL such as Athena the user can introduce primitive methods with arbitrarily complex behavior, which is not possible in HOL. DPLs offer powerful pattern matching facilities on terms and propositions. That is not an option in HOL, because HOL terms and propositions are abstract data types; their concrete representation is hidden. (In principle, that could be changed with a technique similar to Haskell's "views" [13], although implementing such a feature in HOL would require a major effort.) Further, as we explained earlier, DPL implementations can produce certificates, which enhance the credibility of the result and reduce the trusted computing base. Although in principle it is possible to instrument an HOL-like system so that it produces certificates, we are not aware of any actual implementations.

In conclusion, we believe that the explicit distinction between proofs and computations and the use of assumption bases result in a cleaner and more usable framework. They also facilitate the rigorous analysis of proofs, paving the way for formal theories of proof equivalence and optimization. Consider questions such as

- When are two proofs equivalent?
- Does this proof—or proof method—use more assumptions than that one?
- Is the scope of this hypothesis properly contained within the scope of that hypothesis?
- When can one proof be plugged inside another one without causing the latter to fail?

A DPL framework is conducive to formulating and tackling such questions with clarity and precision [1].

Conslusions

We have shown how to use a DPL such as Athena for certified computation. We have demonstrated that certified computation can greatly increase the credibility of our results by isolating and minimizing our trusted computing base. And we have argued that DPLs provide several conceptual and practical advantages to other formalisms that could be used for similar forms of deductive programming.

We stress that certified computation does not guarantee that a program will always produce the correct result. It only ensures that if and when a result is obtained, that result will be correct—modulo the logic that specifies what counts as correct. In a sense, a certified computation reasons about itself: it justifies its own results.

We should also stress that deductive programming in general, and our notion of certified computation in particular, is no panacea for software quality. Dividends are paid only if we manage to formulate a theory that has just the right mixture of simplicity and expressiveness, a task that can be more of an art than a science due to the difficult tension between logic and control. The more we strengthen the logic, the less we have to work on the control. Indeed, in the extreme case we might strengthen the logic so much that everything is provable and we end up with an inconsistent theory. In that case the control part is outstandingly simple: we can just go ahead and output whatever goal is given, with randomly chosen bindings for the existentially quantified variables; it will be a theorem, albeit a useless one. Conversely, the more we weaken the logic the greater the effort that has to go into the control. If we become too conservative and only postulate a very small number of simple axioms and rules that are obviously consistent with the intended interpretation, then proof construction will be difficult. Indeed, in the extreme case we may become so conservative as to postulate no axioms and no rules whatsoever, and then deduction will be more than just difficult—it will be impossible.

So we have to be judicious in crafting our theory. The theory must literally be simple enough, but no simpler. The axioms and rules we postulate must be sufficiently rich to enable us to derive interesting results. Ideally we would like them to be complete, entailing *all* statements that are true in the intended interpretation. In practice we may have to contend ourselves with systems that are complete enough, entailing *most* true statements, or at any rate most true statements such as are likely to be encountered in realistic situations. But above all we must ensure soundness: our axioms and rules must never lead us to statements that are false in the intended interpretation. In fact, our axioms and rules must be evidently sound, or, more plainly put, they must be simple: anyone who is familiar with the problem domain should be able to ascertain their validity without too much reflection. A big selling point of our notion of certified computation is a sharp separation between logic and control. The control engine can be arbitrarily complicated (and in fact the chief advantage of a DPL such as Athena is the great versatility with which proofs can be put together); and yet the DPL semantics guarantee that the theorem that comes out at the end is provable solely by virtue of the postulated axioms and rules. It is this guarantee that allows us to put the focus entirely on the logic: if the logic holds up, we can trust the results. Thus the simpler the logic is, the easier it is to trust the results.

We have also discussed the computational cost of certified computation. In general, as the examples illustrated, certified computations will always take longer than regular computations. The certification process can greatly increase the robustness of our code, but it does not come for free. The question is how much longer certified computations will take, and whether the extra confidence we achieve in our results is worth the extra overhead. This depends on the greater context of the application. When correctness is essential, or when proof of correctness is demanded by an untrusting agent, the advantages of deduction will outweigh its overhead.

But even in situations where efficiency is a serious concern, certified computation could perhaps serve as a useful development paradigm: methods can be used essentially as debugging tools during the development cycle, and removed later when the code is getting ready to ship. In other words, a developer can start out by implementing an algorithm as a method. Because of their stringent logical requirements, testing a method is a much more demanding process than testing a function, and likely to reveal more errors. Further, by going through the exercise of structuring a problem as a formal theory we usually gain a different perspective and a deeper understanding of the various requirements. Once a working method has been derived, it can then be transformed into a conventional algorithm by removing the certifying parts. This eliminates the extra computational overhead of certification but retains the advantage of catching logical errors that would otherwise remain undetected. It is a compromise between full-blown certified computation, which achieves reliability but penalizes performance, and conventional computation, which is efficient but offers no correctness guarantees.

Bibliography

[1] K. Arkoudas. Denotational Proof Languages. PhD thesis, MIT, 2000.

[2] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal Of Automated Reasoning*, 11:43–81, 1993.

[3] M. J. Gordon, A. J. Miller, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[4] 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.

[5] J. Herbrand. Sur la Théorie de la Démonstration. In W. Goldfarb, editor, *Logical Writings of Herbrand*. Cambridge University Press, 1971.

[6] P. Hill and J. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[7] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[8] B. Meyer. Eiffel. In P. H. Salus, editor, *Object-Oriented Programming Languages*, volume 1 of *Handbook of Programming Languages*. Macmillan Technical Publishing, 1998.

[9] D. Miller. A Logic Programming Language with Lambda Abstraction, function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[10] G. Nadathur and D. Miller. Higher-order Horn Clauses. *Journal of the ACM*, 37(4):777–814, 1990.

[11] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the 1999 Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[12] B. Stroustrup. *The C++ Programming Language*. AT&T, Florham Park, New Jersey, USA, 2000.

[13] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987.