MODLIB - Specification Module Library

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 3.21 $

Table of Contents

Introduction

Eli provides a library of specification modules. Each module contains a complete set of specifications that solve a single common task in language implementation. If such a task is identified in the design of an application specification, the module is simply added to the application specification and used there. Most modules solve a rather small task. They can be flexibly combined to solve more complex problems.

This document is organized such that modules which contribute to a certain subtask of language implementation are described together in one chapter.

The first section introduces an example that is referred to in the other sections to explain the use of modules in the context of a language implementation task. The second section explains how modules are obtained from the library and are related to users' specifications. The remaining sections describe sublibraries for a certain problem class each.

The section Chapter 10 [OldModules], page 115 is a quick reference that helps to migrate specifications using modules of previous versions of this library.

This document may be used according to different strategies: In order to become familiar with problem solving by using library modules one can start with the running example of section Chapter 1 [Example], page 5, and extend it as described in the subsequent chapters. One can also start with a particular problem in mind and then search the corresponding chapter for a module that could be used to solve it. The document can also be used as a reference manual for the library modules. Then the [Index], page 121 may also be a starting point. In any case it is recommended to read the introduction of a chapter before using one of its modules.

The modules solve their task by providing specification fragments to one or more Eli tools: Some are simply a C module, others contain `.lido`, `.pdl`, `.ptg`, or references to other library modules. The modules are designed such that as few assumptions as possible have to hold in the user's part of the specification where they are applied. E.g. `.lido` fragments supplied by library modules are mapped to the particular grammar of the user's specification by the inheritance construct of LIDO. Modules described in the Chapter 8 [Adt], page 97 section may also be used directly in C modules or in stand-alone C programs.

Most modules in the library are generic, i.e. the names used in a module can be modified by its instantiation. This facility allows to use the same module for different purposes, e.g. a counter for variables and another for procedures, or specify the element type of a generic list type implementation. Instantiations also control the combination of modules, e.g. association of properties to procedures and to variables which might have different name spaces.

This document describes the task that each module solves, its interface, and how it can be applied in a user's specification. The descriptions are illustrated using a running example introduced in the next section (see Chapter 1 [Example], page 5). Complete executable specifications for that running example are available in the directories `$/Name/Examples` and in `$/Type/Examples`. In general that information should be sufficient to use the modules. A look at the module's text may help in special cases.

Users are encouraged to apply this design technique of decomposition into specification modules to their application as well. The structure of complex designs and their maintainability can thus be improved. The resulting modules may be reused in other applications. The techniques needed to instantiate user developed modules are described in Chapter 2 [Instantiation], page 7.

This library will be extended by adding further modules that solve common tasks. Hence, users are encouraged to contact their Eli distributor if a module for a task that might be of general interest is missing, or if a solution for such a task can be contributed.

1 Running Example

In this section we introduce an example for a language implementation task. It is used in the descriptions of the library modules of this document wherever possible. It shall serve to demonstrate the use of the modules in the context of a complete specification. Of course, such examples can only show certain aspects of the modules. They can not demonstrate their complete functionality and the variety and flexibility of possible applications.

Although the example is taken from the area of programming language implementation, similar tasks are to be solved if languages for other purposes than programming, or programming languages with other characteristics are implemented.

Complete executable specifications for this running example are available in the directories `$/Name/Examples` and in `$/Type/Examples`.

For the purpose of this example we assume that a small artificial programming language is to be implemented. The basic constructs of the language are nested blocks, declarations of variables and of names for values, assignment statements and expressions as specified by the following concrete grammar.

The overall program structure is given by

```
Program:        Source.
Source:         Block.
Block:          Compound.
Compound:       'begin' Declaration* Statement* 'end'.
```

Variable declarations specify the types of the declared variables:

```
Declaration:    'var' ObjDecls ';'.
ObjDecls:       ObjDecl // ','.
ObjDecl:        TypeDenoter Ident.
TypeDenoter:    Ident.
```

For the start of this example there is only a set of predefined type identifiers. In the type analysis section the language is extended by introduction of further types and denoters for them.

For the time being there are only three forms of statements:

```
Statement:      Expression ';'.
Statement:      Variable '=' Expression ';'
Statement:      Block.
```

Further statements are added when necessary to explain aspects of module use.

The expression syntax is left incomplete here in order to introduce operators of different precedence levels (see Section 5.8 [Operator], page 75). If one wants to complete this grammar without adding operators one should add the production

```
Expression:     Operand.
```

The basic operands are

```
Operand:        IntNumber.
Operand:        RealNumber.
Operand:        Variable.
Variable:       Ident.
```

The non-literal tokens are defined by

```
Ident:          PASCAL_IDENTIFIER
IntNumber:      PASCAL_INTEGER
RealNumber:     PASCAL_REAL
                PASCAL_COMMENT
```

The above grammar is chosen such that the solution of language implementation tasks of name analysis, type analysis, and translation by using library modules can be demonstrated. It is also prepared to demonstrate the combination of different module uses. The grammar is extended where necessary in the examples of subsequent sections.

2 Instantiation and Use of Modules

To use a specification module, e.g. the `AlgScope` module in the `Name` library, it is instantiated by the line

```
$/Name/AlgScope.gnrc :inst
```

in a `.specs` file. All component specification files of the module are thus included in the set of specifications. A derivation

```
x.specs:allspecs
```

can be used to inspect all the instantiated files.

The module instance obtained by the instantiation command above provides `.lido` symbol computations for a symbol named `IdUseEnv`, among other specifications.

The above instantiation command is sufficient if only one instantiation of the `AlgScope` module is used. Several instantiations of the same module are distinguished by generic instance names supplied as arguments of the instantiation

```
$/Name/AlgScope.gnrc +instance=CtrlVar :inst
```

In this case another instance of the `AlgScope` module is generated having the instance name `CtrlVar`. It may coexist with the unnamed instance created above. The names of its files and of specified entities (symbols, attributes, etc.) are prefixed by the instance identifier, e.g. `CtrlVarIdUseEnv`.

Some modules have a second generic parameter `referto`. It may specialize the module in a second dimension on instantiation: The `AlgScope` module provides compuations for `Key` attributes in `IdUseEnv` contexts that represent applied occurrences of identifiers. In some situations it may be necessary to compute more than one `Key` attribute in an `IdUseEnv` context (if the identifier is bound in different name spaces). Hence, the `referto` of the `AlgScope` module modifies the names of the `Key` attributes. If the `AlgScope` module is instantiated by

```
$/Name/AlgScope.gnrc +instance=CtrlVar +referto=Ctrl :inst
```

it provides computations for the attribute `CtrlVarIdUseEnv.CtrlKey`, among other computations.

Other modules use the `referto` parameter for different purposes, e.g. specifying the element type for a generic stack module.

If any of the two generic parameters `instance` or `referto` is omitted, as in the first two examples, their value is assumed to be the empty string.

If a module is instantiated as described its facilities can be used in certain components of the user's specification: Symbol computations as those provided by the `AlgScope` module are associated to symbols of the user's tree grammar by `.lido` constructs like

```
SYMBOL UseIdent INHERITS IdUseEnv END;
```

Note: The symbols provided by modules are `CLASS` symbols in the sense of LIDO, i. e. they may not be used directly as tree grammar symbols. `INHERITS` constructs like the above are needed to bind their computations to symbols of user's specification. This means avoids accidental coincidence between names of tree grammar symbols and of module roles.

Further `.lido` specifications may be neccessary to supply information to or obtain information from the thus inherited computations.

Modules may also provide C functions (directly or via specifications for other tools), e.g. functions of the environment module in case of the `AlgScope` module. They may be called in computations of the user's `.lido` specification, or in user's C modules if the appropriate header file is included.

The same instantiation mechanism may be applied to include user defined modules:

```
/user/Lib/ModName.gnrc:inst
```

instantiates the module `ModName` of a user library.

Such a module must contain of at least two files , e.g. `/user/Lib/ModName.fw` containing the module's specifications and `/user/Lib/ModName.gnrc`. The latter is an executable shell script that performs the generic module instantiation. It should have the following form:

```
#!/bin/sh
moddir=`expr $0 : '\(.*\)/.*' \| '.'`
```

```
$1 -e "s/|NAME|/$2/g
s/|KEY|/$3/g" "$moddir"/ModName.fw > "$2"ModName.fw
```

The last two lines use `sed` to substitute the `instance` parameter for any occurrence of `|NAME|` in the file `ModName.fw` and the `referto` parameter for any occurrence of `|KEY|`.

3 Name Analysis Library

Languages usually use names to identify objects. An object is created by an explicit or implicit definition and bound to a name. In a certain range of the text occurrences of that name refer to that object. The scope rules of the language determine where that binding holds. For language implementation a unique name (key) is created for each distinct object and associated to identifiers within the scope of that binding. Name analysis is completed by certain checks of relationships between identifier occurrences as required by the language, e.g. existence of a definition for each identifier use, multiple definitions (See Section 4.2 [OccCnt], page 47, see Section 4.4 [Unique], page 50), or identifier use before its definition (see Section 4.3 [SetFirst], page 49).

This library contains a set of modules which can be used to implement the name analysis task according to a large variety of language rules. The results of these modules are used to check required relationships between identifier occurrences and to solve further subtasks of language implementation, such as type analysis or transformation. Solutions of these tasks are supported by modules of other libraries: Chapter 4 [Prop], page 43, Chapter 5 [TypeAnalysis], page 59, Chapter 7 [Output], page 87.

The module support for name analysis is decomposed into subtasks of increasing complexity. They are described in subsections each. There you find three solution variants for Algol-like, C-like scope rules, and C-like scope rules computed bottom-up while the input is read:

The chapter is structured as follows:

`Preconditions`
: Requirements for the user's tree grammar

`Basic Scope Rules`
: Modules for basic scope rules

`Predefined Identifiers`
: Modules that introduce predefined entities

`Joined Ranges`
: Several subtrees form one conceptual range

`Scope Properties`
: Scopes being propagated as object properties

`Inheritance of Scopes`
: Scopes inherited to other scopes

3.1 Tree Grammar Preconditions

Names are usually represented by identifier terminals. Their denotation is determined by a scanner specification. The grammar then has one (or several) terminals representing identifiers, e.g. `Ident`, as in the running example. The encoding of a particular identifier as computed by a scanner processor is available in contexts where identifier terminal occurs.

Identifiers occur in different contexts: Defining and applied occurrences, or different kinds of identifiers (variables, labels, etc.) may be distinguished. Usually the concrete syntax is designed first, and the different computational roles of identifiers are incrementally developed during the design of the `.lido` specification. Hence, it is recommended NOT to make the distinction in the concrete syntax. It should have the terminal `Ident` in any context. It is rather recommended to distinguish them by LIDO `RULE`s.

Our running example has the concrete productions

```
ObjDecl:        TypeDenoter Ident.
TypeDenoter:    Ident.
Variable:       Ident.
```

We distinguish the different roles of identifiers by introducing new symbol names in the corresponding LIDO `RULE`s:

```
RULE: ObjDecl     ::= TypeDenoter DefIdent END;
RULE: TypeDenoter ::= TypeUseIdent END;
RULE: Variable    ::= UseIdent END;
```

Furthermore, we have to add the necessary chain `RULE`s:

```
RULE: DefIdent     ::= Ident END;
RULE: UseIdent     ::= Ident END;
RULE: TypeUseIdent ::= Ident END;
```

The name analysis modules require that identifier occurrences are represented by nonterminals, like `DefIdent`, `UseIdent`, `TypeUseIdent` as in the example. Each of these symbols has to have an attribute named `Sym` of type `int` representing the identifier encoding. A specification using these modules has to contain suitable computations of the `Sym` attributes. For our example they may be specified like:

```
ATTR Sym: int SYNT;
```

```
SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;

SYMBOL DefIdent     INHERITS IdentOcc END;
SYMBOL UseIdent     INHERITS IdentOcc END;
SYMBOL TypeUseIdent INHERITS IdentOcc END;
```

If your language does not syntactically distinguish between defining and applied identifier occurrences, i.e. objects are introduced by using their name, the above distinction is not necessary. You just introduce `DefIdent` symbols for all occurrences.

Your grammar should have a symbol representing a phrase that contains all (defining and applied) occurrences of a name space. It is usually the root of the whole grammar, e.g. in the running example the symbol `Program`.

If your language has hierarchically nested ranges defining boundaries for the scope of definitions, the abstract syntax should have one or several symbols, e.g. `Range`, `Block`, `Routine`, each representing a range of a name space. If the language does not have nested ranges for definitions you don't need such symbols.

3.2 Basic Scope Rules

The consistent renaming task associates to each identifier occurrence a key that uniquely identifies the object named by the identifier. The following modules solve the basic problems of that task.

Make sure that your tree grammar is constructed according to the advices given in Section 3.1 [Preconditions], page 12.

`AlgScope` Algol-like scope rules

`CScope` C-like scope rules

`BuScope` C-like scope rules analyzed while processing input

Each of the three modules implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey`.

The `AlgScope` module applies Algol-like scope rules:

A binding is valid within the whole smallest range containing the definition, except in inner ranges where a binding for the same identifier holds. I.e. a definition of an `a` in an inner range hides definitions of `a` in outer ranges. An identifier may be used before its definition.

Note: Pascal-like scope rules are a special case of Algol-like scope rules. Additionally it is required that an identifier is not used before its definition. That restriction can be checked using an instance of the `SetFirst` module (see Section 4.3 [SetFirst], page 49).

The `CScope` module applies C-like scope rules:

A binding is valid from the definition upto the end of the smallest range containing the definition, except in inner ranges from a definition of the same identifier to the end of that range. I.e. definitions of `a` in outer range are hidden by a definition of an `a` in an inner range from the point of the definition upto the end of the range. It implies that an identifier is not used before its definition.

The `BuScope` module applies C-like scope rules. Its computations can be executed while the input is read (i.e. while the tree is constructed bottom-up). An application may need this technique if results of the name analysis task influence further reading of input, or results are to be presented to the user while typing the input.

Both Algol-like and C-like scope rules are described by six basic concepts. The modules provide `.lido` specifications with symbol computations for each concept:

`IdDefScope` is a symbol representing a defining identifier occurrence that is bound in the scope of the smallest enclosing range.

`IdUseEnv` is a symbol representing an applied identifier occurrence that is bound in the enclosing environment.

`IdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range.

`ChkIdUse` is a role that may be inherited to an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`RootScope` is the root symbol containing all identifier occurrences and all `RangeScope`. It must be inherited to a grammar symbol. The other symbols may be used depending on the particular scope rule to be specified.

RangeScope is a symbol representing a range for the binding of defining identifier occurrences IdDefScope. It may be nested in RootScope or other ranges.

More details about symbol computations and attributes provided by the three modules are given in the description of each module.

These computational roles are associated to symbols of the user's grammar to solve the basic consistent renaming task.

Complete executable specifications of our running example for each of the three scope rule variants are available in $/Name/Examples/AlgLike.fw, $/Name/Examples/CLike.fw, and $/Name/Examples/BuCLike.fw.

In our running example the roles would be used as follows:

```
SYMBOL Program      INHERITS RootScope END;
SYMBOL Block        INHERITS RangeScope END;
SYMBOL DefIdent     INHERITS IdDefScope END;
SYMBOL UseIdent     INHERITS IdUseEnv END;
SYMBOL TypeUseIdent INHERITS IdUseEnv END;
```

Depending on which of the three modules is instantiated Algol-like or C-like name analysis is performed for this example.

The main result of the task is the computation of the attributes IdDefScope.Key, IdUseEnv.Key, i.e. DefIdent.Key, UseIdent.Key, and TypeUseIdent.Key in our example. They identify the object each identifer is bound to. It may be used in further computations to associate properties to it.

If no binding is found for an applied identifier occurrence the Key attribute has the value NoKey. If that is a violation of language rules an error message can be issued using the role ChkIdUse:

```
SYMBOL UseIdent INHERITS ChkIdUse END;
```

Although both Algol-like and C-like scope rules are defined for nested ranges, the modules may be used for languages that do not have nested ranges, i.e. there is only one single flat range in which definitions are valid. In such a case RootScope is used for that range, and RangeScope is not used.

Another variant of these scope rules arises if a language does not distinguish between defining and applied identifier occurrences: identifiers are defined implicitly by their occurrences. In that case `IdDefScope` is used for that kind of occurrences, and `IdUseEnv` is not used. Of course, this concept does not make sense in languages that have ranges: One could not refer to an outer identifier definition from within an inner range.

We extend our running example to show implicit definitions within a flat range. We add a new kind of variables, say control variables to the language. They are implicitly defined by their use in special statements or operands, given by the following concrete productions:

```
Statement:      'set' Ident 'to' Expression ';'.
Operand:        'use' Ident.
```

These control variable identifiers are bound in a new name space separate from that of the other entities. Hence, we use a second instance of one of the modules. That instance is identified by the generic `instance` parameter `CtrlVar`:

```
$/Name/AlgScope.gnrc +instance=CtrlVar :inst
```

There is only one kind of occurrences for these variables, `CtrlVarUse`, which has the role `CtrlVarIdDefScope`. The `Program` symbol has the role `CtrlVarRootScope`:

```
RULE:  Statement  ::= 'set' CtrlVarUse 'to' Expression ';' END;
RULE:  Expression ::= 'use' CtrlVarUse END;
SYMBOL CtrlVarUse INHERITS CtrlVarIdDefScope, IdentOcc END;
SYMBOL Program    INHERITS CtrlVarRootScope END;
```

3.3 Algol-like Basic Scope Rules

This module implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to Algol-like scope rules:

A binding is valid within the whole smallest range containing the definition, except in inner ranges where a binding for the same identifier holds.

Make sure that you have considered the advices given in Section 3.2 [Basic Scope Rules], page 13.

The module is instantiated by

```
$/Name/AlgScope.gnrc+instance=NAME +referto=KEY :inst
```

Both generic parameters can be omitted in most of the usual applications. The `instance` parameter is used to distinguish several instances of this module. The scope rules of a language may require that identifiers are bound in different name spaces that do not affect each other. Then for each name space an instance of this or of the other basic scope modules is used. The `referto` parameter modifies the names of `Key` attributes. It is only used if there is an identifier occurrence in the language that is bound in more than one name space. These bindings are then described by one `Key` attribute each.

The module provides computational roles for the symbols `NAMERootScope`, `NAMERangeScope`, `NAMEAnyScope`, `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, and `NAMEChkIdUse` to be used in `.lido` specifications. The computations of the module use functions of the library's environment module.

`NAMEIdDefScope` is a symbol representing a defining identifier occurrence.

`NAMEIdUseEnv` is a symbol representing an applied identifier occurrence.

`NAMEIdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range. The outer environment of this range is not considered.

`NAMEChkIdUse` is a role that may be inherited to an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`NAMERootScope` is the root symbol containing all identifier occurrences and all `NAMERangeScope`. It must be inherited to a suitable grammar symbol, usually the root of the grammar.

`NAMERangeScope` is a symbol representing a range for the binding of defining identifier occurrences `NAMEIdDefScope`. It may be nested in `NAMERootScope` or in other ranges.

`NAMEAnyScope` comprises the roles of `NAMERootScope` and `NAMERangeScope`. It may be used in constructs like

```
INCLUDING NAMEAnyScope.NAMEGotKeys
```

The main results of using this module are the bindings of identifier occurrences represented by the attibutes `NAMEIdDefScope.KEYKey` and `NAMEIdUseEnv.KEYKey`.

Usually both `NAMEIdDefScope` and `NAMEIdUseEnv` are used. In specific cases of language rules any combination of `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope` may be used.

The attributes `NAMEIdDefScope.Sym`, `NAMEIdUseEnv.Sym`, `NAMEIdUseScope.Sym` must represent the identifier encoding.

`NAMERootScope.NAMEEnv` is a root environment where all environments of this name space are embedded in.

It has the value of a global variable `NAMERootEnv` that is assigned in the initialization phase of the processor. It allows to introduce predefinitions by initialization code, which then must include the file `NAMERootEnv.h`. (see Section 3.6 [Predefined Identifiers], page 22)

`NAMERangeScope.NAMEEnv` is an inherited attribute for the environment of bindings of this range.

`NAMEAnyScope.NAMEGotKeys` indicates that all keys defined in this and in all enclosing ranges are defined in `NAMEAnyScope.NAMEEnv`. `NAMEAnyScope.NAMEGotKeys` is a precondition for finding a binding using `IdUseEnv`.

`NAMEAnyScope.NAMEGotLocKeys` indicates that all keys are defined in this range are in `NAMEAnyScope.NAMEEnv`.

3.4 C-like Basic Scope Rules

This module implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to C-like scope rules:

A binding is valid from the definition upto the end of the smallest range containing the definition, except in inner ranges from a definition of the same identifier to the end of that range.

Make sure that you have considered the advices given in Section 3.2 [Basic Scope Rules], page 13.

The module is instantiated by

```
$/Name/CScope.gnrc+instance=NAME +referto=KEY :inst
```

Both generic parameters can be omitted in most of the usual applications. The `instance` parameter is used to distinguish several instances of this module. The scope rules of a language may require that identifiers are bound in different name spaces that do not affect each other. Then for each name space an instance of this or of the other basic scope modules is used. The `referto` parameter modifies the names of `Key` attributes. It is only used if there is an identifier occurrence in the language that is bound in more than one name space. These bindings are then described by one `Key` attribute each.

The module provides computational roles for the symbols `NAMERootScope`, `NAMERangeScope`, `NAMEAnyScope`, `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, `NAMEChkIdUse`, and `NAMEIdDefUse` to be used in `.lido` specifications. The computations of the module use functions of the library's environment module.

`NAMEIdDefScope` is a symbol representing a defining identifier occurrence.

`NAMEIdUseEnv` is a symbol representing an applied identifier occurrence.

`NAMEChkIdUse` is a role that may be inherited to an applied identifier occurrence. It issues an error message `identifier is not defined:` if no definition is bound to that identifier.

`NAMEIdUseScope` is a symbol representing an applied identifier occurrence that is bound in the scope of the smallest enclosing range. The outer environment of this range is not considered.

`NAMEIdDefUse` represents a defining identifier occurrence like `NAMEIdDefScope` if `INH.NAMEDefCond` is non-zero, otherwise an applied occurrence like `NAMEIdUseEnv`. `NAMEDefCond` is to be computed by an upper computation. There is a default computation provided that sets `INH.NAMEDefCond` to 1 iff the identifier is not yet bound in the current environment.

`NAMERootScope` is the root symbol containing all identifier occurrences and all `NAMERangeScope`. It must be inherited to a suitable grammar symbol, usually the root of the grammar.

`NAMERangeScope` is a symbol representing a range for the binding of defining identifier occurrences `NAMEIdDefScope`. It may be nested in `NAMERootScope` or other ranges.

`NAMEAnyScope` comprises the roles of `NAMERootScope` and `NAMERangeScope`. It may be used in constructs like

```
INCLUDING NAMEAnyScope.NAMEEnv
```

The main results of using this module are the bindings of identifier occurrences represented by the attibutes `NAMEIdDefScope.KEYKey` and `NAMEIdUseEnv.KEYKey`.

Usually both `NAMEIdDefScope` and `NAMEIdUseEnv` are used. In specific cases of language rules any combination of `NAMEIdDefScope`, `NAMEIdUseEnv`, `NAMEIdUseScope`, `NAMEIdDefUse` may be used.

The attributes `NAMEIdDefScope.Sym`, `NAMEIdUseEnv.Sym`, `NAMEIdUseScope.Sym`, `NAMEIdDefUse.Sym` must represent the identifier encoding.

`NAMERootScope.NAMEEnv` is a root environment where all environments of this name space are embedded in.

It has the value of a global variable `NAMERootEnv` that is assigned in the initialization phase of the processor. It allows to introduce predefinitions by initialization code, which then must include the file `NAMERootEnv.h`. (see Section 3.6 [Predefined Identifiers], page 22)

`NAMERangeScope.NAMEEnv` is an inherited attribute for the environment of bindings of this range.

`NAMEAnyScope.NAMEGotKeys` indicates that all identifer ocurrences from the begin of the `NAMERootScope` upto the end of this range are bound to keys in `NAMEAnyScope.NAMEEnv`.

3.5 C-like Basic Scope Rules Computed Bottom-Up

This module implements consistent renaming of identifiers. The computations of this module are specified such that they are executed while the input program is read. Identifier occurrences are bound to object keys of type `DefTableKey` according to C-like scope rules.

Make sure that you have considered the advices given in Section 3.2 [Basic Scope Rules], page 13.

The module is instantiated by

```
$/Name/BuScope.gnrc+instance=NAME +referto=KEY :inst
```

The functionality provided by this modules is almost the same as that of the `CScope` module (see Section 3.4 [CScope], page 18). Only the differences are described here.

During the bottom-up computation phase values can not be propagated using inherited (`INH`) attributes; and computations that affect a whole subtree, like creation of the scope for a range, have to be associated to a symbol node that precedes that subtree. For that purpose this module provides additional computational roles that cooperate with the usual basic name analysis roles.

Usually it is necessary to introduce additional symbols into the concrete grammar preceding range subtrees. They derive to nothing, and are used to carry the specific bottom-up computations.

In our running example we would replace the productions

```
Source:         Block.
Statement:      Block.
```

by the productions

```
Source:         BuBlock Block.
Statement:      BuBlock Block.
```

and add

```
SYMBOL BuBlock INHERITS CreateNewScope, OpenNewScope END;
```

to the LIDO specification. All other module roles can be used as described for C-like scope rules.

Usually each symbol representing a NAMERangeScope has to be preceded by a symbol that inherits both roles NAMECreateNewScope and NAMEOpenNewScope.

If scopes are used as properties of objects it may be necessary to inherit the roles NAME-CreateNewScope, NAMERecentNewScope, and NAMEOpenNewScope to different symbols which precede a symbol representing a NAMERangeScope. (see Section 3.14 [BuScopeProp], page 33, see Section 3.18 [BuInh], page 40)

`NAMECreateNewScope` creates a new scope that is embedded in the scope of the smallest enclosing range. That scope can be obtained from the attribute `SYNT.NAMENewScope`, or be accessed by a subsequent role `NAMERecentNewScope` (see below).

`NAMEOpenNewScope` makes the scope obtained from `SYNT.NAMENewScope` become the current scope. The attribute `SYNT.NAMEOpenPrecond` can be used to specify a precondition for this operation. If `NAMEOpenNewScope` is inherited to symbol representing an identifier occurrence `SYNT.NAMEOpenPrecond = THIS.KEYKey` ensures that the identifier is bound before the new scope is opened.

`NAMERecentNewScope` accesses the most recently created new scope and provides it by the attribute `SYNT.NAMENewScope`. This role is used together with `NAMEOpenNewScope` if `NAMECreateNewScope` is inherited to a preceding symbol.

We demonstrate the use of these roles for our running example. The grammar introduced in See Chapter 1 [Example], page 5 has to be modified in order to allow bottom-up computation. A new symbol `BuBlock` is introduced. It derives to nothing and precedes the symbol `Block` on right-hand sides of productions:

```
Source:        BuBlock Block.
Statement:     BuBlock Block.
```

The roles `CreateNewScope` and `OpenNewScope` introduce the scope for the subsequent `Block`:

```
SYMBOL Program  INHERITS RootScope END;
SYMBOL Block    INHERITS RangeScope END;
SYMBOL BuBlock  INHERITS CreateNewScope, OpenNewScope END;
```

The other roles for basic scope rules are used as described in See Section 3.4 [CScope], page 18.

3.6 Predefined Identifiers

In most language some identifiers are predefined, e.g. names for basic types or for constants like `true` and `false`. Their definitions are valid in any program as if they were bound in the outermost environment. The two modules `PreDefine` and `PreDefId` described here allow to easily introduce such predefinitions. They require that one of the basic scope rule modules (see Section 3.2 [Basic Scope Rules], page 13) is used.

One single instance of the `PreDefine` module, and an instance of the `PreDefId` for each predefined identifier are needed.

The `PreDefine` module is instantiated by

```
$/Name/PreDefine.gnrc +instance=NAME +referto=IDENT :inst
```

The optional `instance` parameter characterizes the name space in which identifiers are to be predefined. The `instance` parameter has to be the same as that of the basic scope rule module instance used for that name space. Several instances of this module may address different name spaces.

The `referto` parameter specifies the symbol name used for identifier terminals in the grammar. The `referto` parameter must not be omitted.

If a grammar has several identifier terminal symbols predefinitions can be made using several instances of this module, if they belong to different name spaces.

The module provides two functions `NAMEPreDefine` and `NAMEPreDefineSym` which are called by the instance of the `PreDefId` module. `NAMEPreDefineSym` inserts a string into the identifier module to be used as an `IDENT` symbol. `NAMEPreDefine` additionally binds that symbol to a key in the root environment given by the global variable `NAMERootEnv`.

The predefined identifiers are to be described in a file as explained below. The name of that file has to be given as `referto` parameter of the instantiation of the `PreDefId` module:

```
$/Name/PreDefId.gnrc +instance=NAME +referto=(FILENAME) :inst
```

The `instance` parameter has to be the same as that of the `PreDefine` instance. If this instantiation is contained in a `.specs` file and if the description file, say `Predef.d` is contained in the same directory, it may read

```
$/Name/PreDefId.gnrc +referto=(Predef.d) :inst
```

If this instantiation is contained in a `.fw` file, say `AlgLike.fw` it may read

```
$/Name/PreDefId.gnrc +referto=(AlgLike.fw:fwGen/Predef.d) :inst
```

The description file contains a sequence of macro calls, one for each predefined identifier, e.g.

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
PreDefSym ("external", externSym)
PreDefSymKey ("fail", failSym, failKey)
```

`PreDefSym ("xxx", s)` introduces the variable `s`. Its value is the symbol encoding of `"xxx"`.

`PreDefKey ("xxx", k)` introduces the PDL known key `k`; it is bound to the symbol `"xxx"` in the environment `NAMERootEnv`.

`PreDefSymKey ("xxx", s, k)` introduces both `s` and `k` and binds `k` to `s` in the environment `NAMERootEnv`.

The thus introduced variables and known keys may be used in `.lido` specifications; the known keys may be additionally used in any specification where PDL defined entities are available.

Note: The strings specified to be predefined need not obey the rules specified for the notation of `IDENT` symbols.

In our running example we introduce predefined names for some basic types and for boolean constants by the module instantiations

```
$/Name/PreDefine.gnrc +referto=Ident :inst
$/Name/PreDefId.gnrc +referto=(AlgLike.fw:fwGen/Predef.d):inst
```

The file `Predef.d` contains

```
PreDefKey ("int", intKey)
PreDefKey ("real", realKey)
PreDefKey ("bool", boolKey)
PreDefKey ("true", trueKey)
PreDefKey ("false", falseKey)
```

Then key names like `intKey` can be used e.g. in computations for type checking (see Chapter 5 [TypeAnalysis], page 59). In that case it is necessary to state that `true` and `false` are of type bool in a `.pdl` specification:

```
trueKey -> TypeOf = {boolType};
falseKey -> TypeOf = {boolType};
```

It associates the `TypeOf` property to the predefined objects.

3.7 Joined Ranges

In some situations it is not possible to specify the tree grammar such that each range in the sense of scope rules is rooted by one single grammar symbol as required for using the role `RangeScope` of the basic scope module. The following three modules extend the basic scope rule modules (see Section 3.2 [Basic Scope Rules], page 13) by facilities that support such cases:

`AlgRangeSeq`
: Joined Ranges Algol-like

`CRangeSeq`
: Joined Ranges C-like

`BuRangeSeq`
: Joined Ranges C-like Bottom-up

Using one of these modules requires that the corresponding basic scope rule module is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

This module implements the following concept: Several ranges in the program form one single range in the sense of scope rules, i.e. the definition in these ranges contribute to a single scope in which the applied identifier occurrences of these ranges are bound. There is a symbol which is the subtree root for all these ranges. But it is not a range in the sense of scope rules because it may also contain identifier occurrences that are bound in the enclosing range.

The modules provide `.lido` specifications for the following computational roles:

`NAMERangeSequence` is to be inherited to a symbol that is the root of a subtree which contains all to be joined ranges. It provides all attributes provided by `NAMERangeScope`, but it is not a range in the sense of scope rules. Other range roles may not be inerited to `NAMERangeSequence`.

`NAMERangeElement` is the role to be inherited to symbols that represent to be joined ranges. It is a specialized `NAMERangeScope`. Other range roles may not be inherited to such a symbol.

It must be contained in a `NAMERangeSequence` subtree without having a `NAMERangeScope` inbetween.

We demonstrate the use of these facilities by extending the language of our running example by introducing an artificial language construct: It consists of a defining identifier occurrence, that is to be bound in the enclosing range, and two compound statements which form one single range in the sense of scope rules:

```
Statement:      Join.
Join:           'join' DefIdent JoinedBlock JoinedBlock
                'joined' ';'.
JoinedBlock:    Compound.
```

Hence the `Join` symbol has the role `RangeSequence`, and the `JoinedBlock` symbol has the role `RangeElement`:

```
RULE: Join ::= 'join' DefIdent JoinedBlock JoinedBlock
               'joined' ';'
END;

SYMBOL Join INHERITS RangeSequence END;
SYMBOL JoinedBlock INHERITS RangeElement END;
```

This example is applicable with either the `AlgRangeSeq` module or the `CRangeSeq`. In case of Algol-like scope rules an applied identifier occurrence in either of the two ranges may be bound to a definition in either of the two ranges. In case of C-like scope rules an applied identifier occurrence in the second of the two ranges may be bound to a definition in either of the two ranges.

In case of bottom-up computations using the `BuRangeSeq` module some modifications have to be applied as described for that module.

3.8 Joined Ranges Algol-like

This module implements joined ranges for Algol-like scope rules as described in (see Section 3.7 [Joined Ranges], page 25).

The module is instantiated by

```
$/Name/AlgRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `AlgScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence` and `NAMERangeElement` as described in See Section 3.7 [Joined Ranges], page 25.

3.9 Joined Ranges C-like

This module implements joined ranges for C-like scope rules as described in (see Section 3.7 [Joined Ranges], page 25).

The module is instantiated by

```
$/Name/CRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `CScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence` and `NAMERangeElement` as described in See Section 3.7 [Joined Ranges], page 25.

3.10 Joined Ranges C-like Bottom-up

This module implements joined ranges for C-like scope rules as described in (see Section 3.7 [Joined Ranges], page 25). Its computations are executed `BOTTOMUP` while the input is read.

The module is instantiated by

```
$/Name/BuRangeSeq.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `BuScope` is instantiated with the same values of the generic parameters.

The module provides the computational roles `NAMERangeSequence`, `NAMEOpenSeqScope` and `NAMEOpenElemScope`.

`NAMERangeSequence` is to be inherited to a symbol that is the root of a subtree which contains all to be joined ranges. It provides all attributes provided by `NAMERangeScope`, but it is not a range in the sense of scope rules. Other range roles may not be inerited to `NAMERangeSequence`.

`NAMEOpenSeqScope` is to be inherited to a symbol that is in the subtree of `NAMERangeSequence` and precedes all to be joined ranges.

`NAMEOpenElemScope` is to be inherited to a symbol that precedes each to be joined range.

A nested `NAMERangeSequence` may not occur between `NAMEOpenSeqScope` and the first `NAMEOpenElemScope`.

The example given in See Section 3.7 [Joined Ranges], page 25 is here modified for the bottom-up case.

We demonstrate the use of these facilities by extending the language of our running example by introducing an artificial language construct: It consists of a defining identifier occurrence, that is to be bound in the enclosing range, and two compound statements which form one single range in the sense of scope rules.

We introduce two new symbols `BuJoin` and `BuJoinEl` that derive to empty.

```
Statement:      Join.
Join:           'join' BuJoin DefIdent
                       BuJoinEl Block BuJoinEl Block
                'joined' ';'.
BuJoin:         .
BuJoinEl:       .
```

The module roles are inherited as described above:

```
RULE: Join ::= 'join' BuJoin DefIdent
                      BuJoinEl Block BuJoinEl Block
               'joined' ';'
END;

SYMBOL Join INHERITS RangeSequence END;
SYMBOL BuJoin INHERITS OpenSeqScope END;
SYMBOL BuJoinEl INHERITS OpenElemScope END;
SYMBOL Block INHERITS RangeScope END;
```

3.11 Scopes Being Properties of Objects

Language constructs like modules, classes, or record types have a body that is a range. The set of bindings for the components defined in that range constitutes its scope. In an applied context of a module, class, or record identifier its components may be selected, e.g. in `m.k`, where `m` is a module identifier and `k` is one of its components. In order to bind such applied occurrences of component identifiers in contexts outside their defining range, the scope of the range is associated as a property to the key of the module, class, or record identifier.

This specific task of consistent renaming for component identifiers is closely related to type analysis. If in `v.k` is a variable that has a record type, then that type key has the scope of the record range associated as a property (see Chapter 5 [TypeAnalysis], page 59).

The following three modules extend the basic scope rule modules (see Section 3.2 [Basic Scope Rules], page 13) by facilities that support scope properties:

`AlgScopeProp`
: Scope Properties Algol-like

`CScopeProp`
: Scope Properties C-like

`BuScopeProp`
: Scope Properties C-like analyzed while processing input

Using one of these modules requires that the corresponding basic scope rule module is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

Each of the modules introduces a `PDL` property named `NAMEScope` where `NAME` is the value of the `instance` parameter.

The modules provide `.lido` specifications for the following computational roles:

`NAMERangeScopeProp` is a range. Its scope is to be associated as the `NAMEScope` property to the key `THIS.KEYScopeKey`. An upper or lower computation of `THIS.KEYScopeKey` has to be provided. All local definitions are bound in this scope.

`NAMEIdUseScopeProp` is an applied identifier ocurrence that is bound within the scope given by the attribute `THIS.NAMEScope`. Its computation may be provided in the upper or the lower context.

The attribute `NAMERootScope.GotScopeProp` represents the condition that the scope properties are associated thoughout the program. It may be used as a precondition for computations that access the property.

We demonstrate the use of these facilities by extending the language of our running example by module declarations and access of module components. The notation is specified by the following two concrete productions:

```
Declaration:    'module' DefIdent ModBlock ';'.
ModBlock:       Compound.
Operand:        ModUseIdent '::' QualIdent.
ModUseIdent:    Ident.
QualIdent:      Ident.
```

The symbols inherit the roles provided by the scope property module as described above:

```
RULE: Declaration ::= 'module' DefIdent ModBlock ';' COMPUTE
      ModBlock.ScopeKey = DefIdent.Key;
END;

SYMBOL ModBlock INHERITS RangeScopeProp END;
```

In the context of the module declaration it is specified that the scope of the module body is associated to the key of the module identifer.

In the context of the selection the scope is specified in which the selected component is to be bound. It is accessed from the key of the module identifer. The dependency ensures that all scope properties are associated before accessed here:

```
RULE: Expression  ::= ModUseIdent '::' QualIdent COMPUTE
      QualIdent.Scope = GetScope (ModUseIdent.Key, NoEnv)
          <- INCLUDING RootScope.GotScopeProp;
END;

SYMBOL ModUseIdent INHERITS
       IdUseEnv, ChkIdUse, IdentOcc
END;

SYMBOL QualIdent   INHERITS
       IdUseScopeProp, ChkIdUse, IdentOcc
END;
```

In order to make sure that it is really a module identifier to which the selection is applied we specify the following check

```
SYMBOL ModUseIdent INHERITS ChkModUseIdent END;

CLASS SYMBOL ChkModUseIdent COMPUTE
  IF (AND (NE (THIS.Key, NoKey),
           EQ (GetScope (THIS.Key, NoEnv), NoEnv)),
  message (FATAL, CatStrInd ("module identifier required: ",
                             THIS.Sym),
           0, COORDREF))
  <- INCLUDING RootScope.GotScopeProp;
END;
```

The check is associated to a role `ChkModUseIdent` for being reused in other contexts. The message is only issued if the identifier is defined but does not have a scope property.

(The `Strings` module is used to compose the message text (see Section 9.2 [Strings], page 111).)

The described example is applicable with either the `AlgScopeProp` module or the `CScopeProp`. (The precondition `INCLUDING NAMERootScope.GotScopeProp` should not be used in case of `CScopeProp`.) In case of bottom-up computations using the `BuScopeProp` module some modifications have to be applied as described for that module.

3.12 Scope Properties Algol-like

This module implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that all scope properties are associated before any is accessed. This strategy fits to Algol-like scope rules.

The module is instantiated by

```
$/Name/AlgScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `AlgScope` is instantiated with the same values of the generic parameters.

The module provides a PDL property named NAMEScope and the computational roles NAMERangeScopeProp and NAMEIdUseScopeProp, as described in (see Section 3.11 [Scope Properties], page 29).

If a call of GetNAMEScope in a user computation accesses a scope property that computation has to depend on INCLUDING NAMERootScope.GotScopeProp. It guarantees that all scope properties are set and all their components are defined.

3.13 Scope Properties C-like

This module implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that scope properties are associated and accessed in left-to-right depth-first order. This strategy fits to C-like scope rules.

The module is instantiated by

```
$/Name/CScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module CScope is instantiated with the same values of the generic parameters.

The module provides a PDL property named NAMEScope and the computational roles NAMERangeScopeProp and NAMEIdUseScopeProp, as described in (see Section 3.11 [Scope Properties], page 29).

All computations of this module follow strictly C-like scope rules, i.e. binding of identifier occurrences, association of scope properties, and access of scope properties are done in left-to-right depth-first order.

Calls of GetNAMEScope in a user computation do not need a specific precondition if their result is used in the module role NAMEIdUseScopeProp. Only if a particular computation is to depend on the fact that all scope properties of the program are associated, it may depend on INCLUDING NAMERootScope.GotScopeProp.

3.14 Scope Properties C-like Bottom-Up

This module implements consistent renaming of identifiers using scopes which are properties associated to object keys. The module computations ensure that scope properties are associated and accessed in left-to-right depth-first order. This strategy fits to C-like scope rules.

The computations provided by this module are executed while reading the input.

The module is instantiated by

```
$/Name/BuScopeProp.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the module `BuScope` is instantiated with the same values of the generic parameters.

The module provides a `PDL` property named `NAMEScope` and the computational roles `NAMEIdSetScopeProp`, `NAMEIdGetScopeProp`, and `NAMEIdUseScopeProp`. A role `NAMERangeScopeProp` is NOT provided; `NAMERangeScope` has to be used instead.

All computations of this module follow strictly C-like scope rules, i.e. binding of identifier occurrences, association of scope properties, and access of scope properties are done in left-to-right depth-first order.

As a consequence of bottom-up computation the value of a key can not be propagated by an upper computation to the range symbol. Hence, if the defining identifier occurrence precedes the range, the scope has to be created by the role `NAMECreateNewScope` (see Section 3.5 [BuScope], page 20) and associated to the key in the identifier context using the role `NAMEIdSetScopeProp`.

The role that opens the range scope (`NAMEOpenNewScope`, see Section 3.5 [BuScope], page 20) may also be associated to that identifier context, avoiding an additional symbol that derives to empty.

The range symbol has the role `NAMERangeScope`.

The module declaration of our example then reads:

```
RULE: Declaration ::= 'module' ModDefIdent Block ';' END;

SYMBOL ModDefIdent INHERITS
       CreateNewScope, OpenNewScope, IdSetScopeProp,
       IdDefScope, IdentOcc
```

```
COMPUTE
  SYNT.OpenPrecond = SYNT.Key;
END;
```

`NAMEOpenPrecond` is specified to depend on the key attribute to ensure that the identifier is bound in the enclosing environment before the environment of the module range is opened.

In component selections the scope property needs to be propagated from the context that provides it to the selector context. The module role `NAMEGetScopeProp` accesses the scope from the key specified by `KEYScopeKey` and assigns it to a variable. It is used at the selector context right of it by the role `NAMEIdUseScopeProp`.

Hence, in our running example the selection is specified as follows:

```
RULE: Expression  ::= ModUseIdent '::' QualIdent END;

SYMBOL ModUseIdent INHERITS
       GetScopeProp, IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = THIS.Key;
END;

SYMBOL QualIdent   INHERITS
       IdUseScopeProp, ChkIdUse, IdentOcc
END;
```

If we had a typed record expression instead of the module identifier to select from, `ScopeKey` would be set to the type key instead of the module key.

3.15 Inheritance of Scopes

The basic scope rule concepts are described by hierarchically nested environments which reflect the structure of nested ranges in a program. Using scopes as properties of objects, as described in See Section 3.11 [Scope Properties], page 29 allows to bind single identifiers that occur outside of the range which forms the scope property, e.g. a component identifier that is qualified by a module name.

In this section we further extend that concept such that scope rules for language constructs like `with` statements of Pascal, `use` qualifications of Ada, or inheritance of classes as in object-oriented languages can be specified.

The new concept is described by an inheritance relation between scopes that is used when applied identifier occurrences in a range are bound to definitions.

There are three modules that provide computations for the consistent renaming task based on inheritance. They rely on the use of the corresponding modules for basic scope rules and for scope properties:

`AlgInh` Inheritance with Algol-like Scope Rules

`CInh` Inheritance with C-like Scope Rules

`BuInh` Inheritance computed while processing input

Using one of these modules requires that the corresponding basic scope rule module and the scope property is instantiated with the same generic parameters `+instance=NAME` and `+referto=KEY`.

Each of the three modules implements consistent renaming of identifiers. Identifier occurrences are bound to object keys of type `DefTableKey` according to the following inheritance rule:

An inheritance relation between scopes is introduced: A scope `c1` may inherit the bindings of a scope `c2`, i.e. a definition of `c2` is inherited to `c1` unless it is hidden by another definition of the same identifier in `c1`. A scope may inherit from several scopes (multiple inheritance). The inheritance relation is transitive and must be acyclic.

Together with the nesting of ranges the following general scope rule is applied:

An applied occurrence of an identifier `a` is bound to a definition of `a` which is contained in or inherited to the smallest enclosing range.

Definitions contained in a range hide definitions inherited (directly or indirectly) to that range.

Definitions inherited to a range hide definitions of enclosing ranges.

Using multiple inheritance a scope `c1` may inherit from a scope `c2` and from `c3`, where `c2` also inherits from `c3`. If both `c2` and `c3` define an identifier `a`, then the definition of `a` in `c3` is hidden by that of `c2`. This holds for `c1`, too, although there is an inheritance path from `c3` to `c1` that does not pass `c2`.

If several definitions of an identifier `a` are inherited via different unrelated inheritance paths, the applied occurrence is bound to an arbitrary one of them. This module provides a means to detect that situation, in order to issue an error message of to access all those definitions, depending on the requirements of the particular language.

The modules provide `.lido` specifications for the following computational roles:

`NAMERangeSnglInh` is a range that inherits a single scope. It is obtained from the NAMEScope property of the attribute `THIS.KEYInhKey`. An upper or lower computation for the attribute `THIS.KEYInhKey` has to be provided. The attribute `SYNT.NAMEInheritOk` is set to 1 iff the inheritance relation is legal, i.e. non cyclic and both scopes belong to the same environment hierarchy.

`NAMERangeMulInh` is a range that may inherit several scopes using the role `NAMEInheritScope` (described below) or the function `InheritClass` (provided by the environment module). An upper or lower computation for the `VOID` attribute `THIS.GotInh` has to be provided such that it is the postcondition stating that all those inheritances are done.

`NAMEInheritScope` requires that two attributes of type `Environment`, `THIS.NAMEOuterScope` and `THIS.NAMEInnerScope`. are computed by upper or lower computations. The inheritance relation from `THIS.NAMEOuterScope` to `THIS.NAMEInnerScope` is established, i.e. definitions of `THIS.NAMEOuterScope` are to be inherited to `THIS.NAMEInnerScope`.

The attribute `SYNT.NAMEInheritOk` is set to 1 iff the inheritance relation is legal, i.e. non cyclic and both scopes belong to the same environment hierarchy.

`NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` are roles to be associated to an applied identifier occurrence. They issue a message if several definitions of the identifer are inherited on different unrelated inheritance paths. `NAMEChkInhIdUse` may be used together with `NAMEIdUseEnv` or `NAMEIdUseScope`; `NAMEChkInhIdUseScopeProp` may be used together with `NAMEIdUseScopeProp`.

The modules provide also attributes `NAMEAnyRange.NAMEGotInhScopes` of typ `VOID`. This is a precondition for any consistent renaming that may be affected by an inheritance. It has to be stated in any such user's computation not provided by this module.

We demonstrate the use of single inheritance by extending our running example by a `with` statement for modules (see Section 3.11 [Scope Properties], page 29).

```
Statement:    'with' ModUseIdent 'do' WithBody.
```

```
WithBody:     Statement.
```

The identifier should be bound to a module. The `WithBody` inherits the module's scope. I.e. the definitions of the module body are valid in the `WithBody`. They may be hidden by definitions in ranges contained in the `WithBody`. They may hide definitions in ranges enclosing the `with` statement. Hence, the `WithBody` plays the role of a `RangeSnglInh` and its scope is the target of the inheritance relation. The module's scope property is its source:

```
RULE: Statement ::= 'with' ModUseIdent 'do' WithBody COMPUTE
  WithBody.InhKey = ModUseIdent.Key;
END;

SYMBOL WithBody INHERITS RangeSnglInh END;
```

Similarly we can extend the language of our running example by classes with multiple inheritance:

```
Declaration:   'class' DefIdent Inheritances ClassBlock ';'.
ClassBlock:    Compound.
Inheritances:  Inheritance*.
Inheritance:   ':' InheritIdent.
InheritIdent:  Ident.
```

A declaration of a class resembles that of a module (see Section 3.11 [Scope Properties], page 29). Additionally other classes may be inherited to it, i.e. their definitions are valid within the class body, if not hidden by innner definitions. The inherited definitions may hide definitions in ranges the class declaration is contained in. Hence, the scope of the class body is the target of all `Inheritances`, their sources are given by the scope property associated to the classes identified in the `Inheritances`.

```
SYMBOL ClassBlock INHERITS RangeScopeProp, RangeMulInh END;

RULE: Declaration ::= 'class' DefIdent Inheritances ClassBlock ';'
COMPUTE
  ClassBlock.ScopeKey = DefIdent.Key;
  ClassBlock.GotInh =
     Inheritances CONSTITUENTS InheritIdent.InheritOk;
  Inheritances.ToEnv = ClassBlock.Env;
END;

SYMBOL Inheritances:  ToEnv: Environment;

SYMBOL InheritIdent INHERITS
```

```
      InheritScope, IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.InnerScope = INCLUDING Inheritances.ToEnv;
  SYNT.OuterScope = GetScope (THIS.Key, NoEnv)
        <- INCLUDING RootScope.GotScopeProp;

  IF (NOT (THIS.InheritOk),
  message (ERROR, CatStrInd ("wrong inheritance: ", THIS.Sym),
           0, COORDREF));
END;
```

Languages (like C++) allow that different definitions of an identifier may be inherited on different inheritance paths to a range. But in that case such an identifier may not be applied in that range. This restriction is checkedby the roles `ChkInhIdUse` and `ChkInhIdUseScopeProp`. They have to be associated to applied identifier symbols which are bound in the enclosing environment or in the scope obtained from a property, respectively:

```
SYMBOL UseIdent INHERITS ChkInhIdUse END;
SYMBOL QualIdent INHERITS ChkInhIdUseScopeProp END;
```

The above specification also fits to the specification for identifiers that are qualified by a module name given in (see Section 3.11 [Scope Properties], page 29). If in a construct `c::x` `c` is a class, then `x` is bound to a component defined in `c` or in a class inherited to `c`. This is the concept of the scope operator in C++.

These examples are applied in the same way for Algol-like and for C-like scope rules. The differences for bottom-up computation are explained in the description of the `BuInh` module.

3.16 Inheritance with Algol-like Scope Rules

This module implements consistent renaming of identifiers according to inheritance relations based on Algol-like scope rules as described in See Section 3.15 [Inheritance of Scopes], page 35.

The module is instantiated by

```
$/Name/AlgInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `AlgScope` and `AlgScopeProp` are instantiated with the same values of the generic parameters.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Section 3.15 [Inheritance of Scopes], page 35.

3.17 Inheritance with C-like Scope Rules

This module implements consistent renaming of identifiers according to inheritance relations based on C-like scope rules as described in See Section 3.15 [Inheritance of Scopes], page 35.

The module is instantiated by

```
$/Name/CInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `CScope` and `CScopeProp` are instantiated with the same values of the generic parameters.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Section 3.15 [Inheritance of Scopes], page 35.

3.18 C-like Inheritance Bottom-Up

This module implements consistent renaming of identifiers according to inheritance relations based on C-like scope rules. The computations can be executed while input is read.

The module is instantiated by

```
$/Name/BuInh.gnrc+instance=NAME +referto=KEY :inst
```

Using this module requires that the modules `BuScope` and `BuScopeProp` are instantiated with the same values of the generic parameters.

The module provides `.lido` specifications for the computational roles `NAMERangeSnglInh`, `NAMERangeMulInh`, `NAMEInheritScope`, `NAMEChkInhIdUse` and `NAMEChkInhIdUseScopeProp` as described in See Section 3.15 [Inheritance of Scopes], page 35. No additional range role is provided by this module. The role `NAMERangeScope` of the basic scope rule module is to be used for ranges that are affected by inheritance, too.

The role `NAMEInheritScope` differs from the description in See Section 3.15 [Inheritance of Scopes], page 35:

The target scope for the inheritance relation is assumed to be computed by the role `NAMECreateNewScope` in this context or in a preceding context. It is passed via a variable. If `NAMECreateNewScope` and `NAMEInheritScope` are used in the same context, a computation `SYNT.NAMEInhPrecond = THIS.NAMENewScope;` has to be added, in order to guarantee proper use of the variable.

A lower computation of `SYNT.NAMEOuterScope` is required for this context.

The examples given in See Section 3.15 [Inheritance of Scopes], page 35 are modified here to allow for bottom-up computation using this module.

We demonstrate the use of single inheritance by extending our running example by a `with` statement for modules (see Section 3.11 [Scope Properties], page 29).

```
Statement:    'with' WithUseIdent 'do' WithBody.
WithBody:     Statement.
```

The identifier should be bound to a module. The `WithBody` inherits the module's scope. I.e. the definitions of the module body are valid in the `WithBody`. They may be hidden by definitions in ranges contained in the `WithBody`. They may hide definitions in ranges enclosing the `with` statement. `WithBody` plays the role of a `RangeScope`. In the preceding `WithUseIdent` context the scope is created and determined to be target of an inheritance relation. The scope property of the module key is stated to be the outer scope of the inheritance relation.

```
RULE: Statement ::= 'with' WithUseIdent 'do' WithBody END;

SYMBOL WithBody INHERITS RangeScope END;
SYMBOL WithUseIdent INHERITS
       GetScopeProp, CreateNewScope, InheritScope,
```

```
       OpenNewScope, IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = SYNT.Key;
  SYNT.OuterScope = SYNT.ScopeProp;
  SYNT.OpenPrecond = SYNT.Key;
END;
```

Similarly we can extend the language of our running example by classes with multiple inheritance:

The scope of the class body is created in the context `ClassDefIdent`, associated a property of the class identifier, and used as a target for the inheritance relations established in all `Inheritances`. The roles `RecentNewScope` and `OpenNewScope` in the newly introduced context `BuClass` access and open that scope.

```
RULE: Declaration ::= 'class' ClassDefIdent Inheritances
                              BuClass ClassBlock ';'
END;

SYMBOL ClassDefIdent INHERITS
       CreateNewScope, IdSetScopeProp, IdDefScope, IdentOcc
END;

SYMBOL BuClass INHERITS RecentNewScope, OpenNewScope END;
```

In the `InheritIdent` contexts the scope property of the identifer is accessed and determined to be the outer scope to be inherited to the prevoiusly created scope.

```
SYMBOL InheritIdent INHERITS
       GetScopeProp, InheritScope,
       IdUseEnv, ChkIdUse, IdentOcc
COMPUTE
  SYNT.ScopeKey = SYNT.Key;
  SYNT.OuterScope = SYNT.ScopeProp;

  IF (AND (NOT (THIS.InheritOk), NE (THIS.Key, NoKey)),
  message (FATAL, CatStrInd ("cyclic inheritance: ", THIS.Sym),
           0, COORDREF))
  BOTTOMUP;
END;
```

The above specification also fits to the specification for identifiers that are qualified by a module name given in (see Section 3.11 [Scope Properties], page 29). If in a construct `c::x` `c` is a class, then `x` is bound to a component defined in `c` or in a class inherited to `c`. This is the concept of the scope operator in C++.

4 Property Library

The input for a text processor usually describes objects that have certain properties, e.g. named entities like variables in a program, or fields of a data base. Their properties are determined, used, or checked according to the context in which an object occurs. An object may occur several times in the input. The occurrences are mapped to a unique identification, a key (see Chapter 3 [Name], page 11). Properties are associated and accessed via those keys. Properties are represented by values of certain types. The Eli tool PDL is used to generate functions that store property values in a data base of the language processor and that retrieve values from it.

The first section describes how the modules are instatiated and used, the others describe the modules contained in this library:

`Usage` Common Aspects of Property Modules

`OccCnt` Count Occurrences of Objects

`SetFirst` Set a Property at the First Object Occurrence

`Unique` Check for Unique Object Occurrences

`FirstOcc` Determine First Object Occurrence

`ObjCnt` Map Objects to Integers

`Kind` Associate Kinds to Objects

`KindSet` Associate Sets of Kinds to Objects

`Reflex` Reflexive Relations Between Objects

`PropLib` Some Useful PDL Specifications

The modules of this library can solve a large variety of tasks. E.g. the `OccCnt` module enumerates occurrences of each object. Its result can be used to check whether an identifier is multiply defined. It can also be used to trigger an output operation exactly once for each object. Such an output may say how often the identifier occurs in the program, or it may be a declaration of the identifier in the target code.

4.1 Common Aspects of Property Modules

The use of any module of this library requires that objects are identified by keys as computed by the consistent renaming modules.

All modules of this library, except `SetFirst`, `Reflex`, and `PropLib`, are instantiated by the same pattern:

```
$/Prop/ModuleName.gnrc+instance=NAME +referto=KEY :inst
```

for example

```
$/Prop/OccCnt.gnrc+instance=Var +referto=CtrlVar :inst
```

The `instance` parameter is used to distinguish several instances of one module that are used in one specification. If only one instance of a module is used the parameter can be omitted.

The `referto` parameter is used to specify the name of the `Key` attribute, `CtrlVarKey` in the example above. The value must be the same as that of the `referto` parameter specified for the instance of the consistent renaming module which computed the `Key` attribute. (The `referto` parameter is usually omitted, unless there are symbols that have more than one `Key` attribute.)

The instantiation of the modules `SetFirst`, `Reflex`, and `PropLib` is described in the corresponding section.

All modules of this library, except `PropLib`, require that a grammar symbol is specified which comprises the range of the input where computations of the module may occur. In general the root of the tree grammar can be chosen for that purpose. Each module provides a computational role that has to be associated to that grammar symbol. For a module having the name `ModName` that is instantiated with `+instance=NAME` that role is named `NAMERangeModName`. (Exception: in the module `OccCnt` it is named `NAMERangeCnt`.)

The computations provided by each module ensure that properties are not accessed before they are set within the range. The condition that all properties are set within the range is provided by an attribute of the range symbol. It may be used as a precondition for computations that rely on that fact.

In cases where the computation of a property value in one range depends on the access of a property in another (e.g. enclosing) range, it is necessary to choose smaller ranges than the grammar root. The ranges may also be recursively nested.

Note: The ranges considered here may, but need not correspond to the ranges used in the unique renaming modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

The computations provided by each module ensure that properties are not accessed before they are set within the range. The condition that all properties are set within the range is provided by an attribute of the range symbol. It may be used as a precondition for computations that rely on that fact.

In cases where the computation of a property value in one range depends on the access of a property in another (e.g. enclosing) range, it is necessary to choose smaller ranges than the grammar root. The ranges may also be recursively nested.

Note: The ranges considered here may, but need not correspond to the ranges used in the unique renaming modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

The computations provided by each module ensure that properties are not accessed before they are set within the range. The condition that all properties are set within the range is provided by an attribute of the range symbol. It may be used as a precondition for computations that rely on that fact.

In cases where the computation of a property value in one range depends on the access of a property in another (e.g. enclosing) range, it is necessary to choose smaller ranges than the grammar root. The ranges may also be recursively nested.

Note: The ranges considered here may, but need not correspond to the ranges used in the unique renaming modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

The computations provided by each module ensure that properties are not accessed before they are set within the range. The condition that all properties are set within the range is provided by

an attribute of the range symbol. It may be used as a precondition for computations that rely on that fact.

In cases where the computation of a property value in one range depends on the access of a property in another (e.g. enclosing) range, it is necessary to choose smaller ranges than the grammar root. The ranges may also be recursively nested.

Note: The ranges considered here may, but need not correspond to the ranges used in the unique renaming modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

The computations provided by each module ensure that properties are not accessed before they are set within the range. The condition that all properties are set within the range is provided by an attribute of the range symbol. It may be used as a precondition for computations that rely on that fact.

In cases where the computation of a property value in one range depends on the access of a property in another (e.g. enclosing) range, it is necessary to choose smaller ranges than the grammar root. The ranges may also be recursively nested.

Note: The ranges considered here may, but need not correspond to the ranges used in the unique renaming modules.

The computational results of each module can be accessed using attributes in `.lido` computations, or by application of PDL generated access functions applied to object keys, as described for each module individually.

In our running example we use the `OccCnt` module to check for multiply defined identifiers. As it is the first use of this module we can ommit the `instance` parameter. Since we omitted the `referto` parameter in the instance of the consistent renaming module that computes the keys, we omit it here, too:

```
$/Prop/OccCnt.gnrc :inst
```

The central computations of each module are provided by one or several computational roles, e.g. NAMECount and NAMETotalCnt in case of the OccCnt module. These roles are usually associated to grammar symbols representing identifier occurrences. In general they may be associated to any symbol that has a Key attribute.

In order to check for multiply defined identifiers in our running example both the Count role and the TotalCnt role is associated to defining identifier occurrences. As there are several symbols representing defining identifier occurrences which all have to be checked in the same way, we introduce a new role MultDefChk that comprises the necessary computations:

```
SYMBOL MultDefChk    INHERITS Count, TotalCnt END;

SYMBOL DefIdent      INHERITS MultDefChk END;
SYMBOL ClassDefIdent INHERITS MultDefChk END;
SYMBOL ModDefIdent   INHERITS MultDefChk END;
```

In our running example we associate the role RangeCnt to the grammar root by:

```
SYMBOL Program INHERITS RangeCnt END;
```

4.2 Count Occurrences of Objects

The computations of this module enumerate certain occurrences of objects represented by symbols that have a Key attribute. That are usually certain occurrences of identifiers.

The information computed by the module may be used for different purposes, e.g. for statistics about the input text, checks for unique occurrences, computations at the first or the last occurrence, etc.

The module is instantiated by

```
$/Prop/OccCnt.gnrc+instance=NAME +referto=KEY :inst
```

The module provides two computational roles, NAMECount and NAMETotalCnt, besides the role NAMERangeCnt that specifies the range where these these computations may occur (see Section 4.1 [Usage], page 43). The roles may be associated to one or several grammar symbols.

Let k be a key, then all occurrences of k in a NAMECount context are enumerated in left-to-right depth-first order. The attribute NAMECount.NAMECnt is the number of k's occurrence with respect to this order. The total number of occurrences of `k` is associated as the property `NAMECnt` to `k`. The role `NAMETotalCnt` makes it available as an attribute `NAMETotalCnt.NAMETotalCnt`. If the `NAMECnt` property is accessed directly in user's computations, those have to state `NAMERangeCnt.GotNAMECnt` as precondition.

In (see Section 4.1 [Usage], page 43), we explained how to associate these rule to grammar symbols in order to check for multiply definitions:

```
SYMBOL Program       INHERITS RangeCnt END;
SYMBOL MultDefChk    INHERITS Count, TotalCnt END;

SYMBOL DefIdent      INHERITS MultDefChk END;
SYMBOL ClassDefIdent INHERITS MultDefChk END;
SYMBOL ModDefIdent   INHERITS MultDefChk END;
```

The check is completed by using the results of this module in computations associated to `MultDefChk`:

```
SYMBOL MultDefChk COMPUTE
   IF (GT (THIS.TotalCnt, 1),
   message (ERROR, CatStrInd ("identifier is multiply defined: ", THIS.Sym),
            0, COORDREF));
END;
```

The following demonstrates a different application of this module. Assume we want to print how often each object in a program is referenced in some context. Hence, any identifier occurrence has to be counted. To avoid that this module application collides with the previous, we have to use a different instance of the `OccCnt` module:

```
$/Prop/OccCnt.gnrc +instance=Prnt:inst
```

That is easily achieved by associating the `Count` and the `TotalCnt` roles to the role `IdentOcc` previously introduced in our running example:

```
SYMBOL Program   INHERITS PrntRangeCnt END;
SYMBOL IdentOcc  INHERITS PrntCount, PrntTotalCnt COMPUTE
   IF (EQ (THIS.PrntCnt, 1),
   printf ("identifier %s occurs %d times\n",
           StringTable (THIS.Sym), THIS.PrntTotalCnt));
END;
```

4.3 Set a Property at the First Object Occurrence

This module associates values of type `TYPE` as property `NAME` to objects identified by keys. The property is set at most once at the first occurrence of the object which has the `NAMESetFirst` role. The module may for example be used to associate source coordinates of defining identifier occurrences to objects.

The module instantiation differs from the usual pattern for this library:

```
$/Prop/SetFirst.gnrc+instance=NAME +referto=TYPE :inst
```

Note: This module is not applicable to symbol occurrences that do not have an attribute named `Key`, e.g. due to the use of the `referto` parameter for a consistent renaming module.

Values are associated in the first `NAMESetFirst` context with respect to left-to-right depth-first tree order. The property value to be set has to be provided by a user's computation for the attribute `NAMESetFirst.NAME` (in any, not only the first `NAMESetFirst` context).

The range for these computations has to be specified by the role `NAMERangeSetFirst` (see Section 4.1 [Usage], page 43). The attribute `NAMERangeSetFirst.GotNAME` has to be used as a precondition for computations which access the `NAME` property to guarantee that the property is set.

If we want to associate the source coordinates of defining identifier occurrences to object keys in our running example we instantiate this module using the coordinate type `CoordPtr` exported by the error module, and the property name `DefPt`

```
$/Prop/SetFirst.gnrc+instance=DefPt +referto=CoordPtr :inst
```

Then the roles of this module are associated to our grammar symbols:

```
SYMBOL Program       INHERITS DefPtRangeSetFirst END;
SYMBOL DefPoint      INHERITS DefPtSetFirst COMPUTE
   SYNT.DefPt = COORDREF;
END;

SYMBOL DefIdent      INHERITS DefPoint END;
SYMBOL ClassDefIdent INHERITS DefPoint END;
SYMBOL ModDefIdent   INHERITS DefPoint END;
```

The property computed this way may for example be used to check whether an identifier occurs in an applied context before its definition, as required e.g. in Pascal. For that purpose a function `CoordLess` is used to compare coordinates. (It is provided by the `PropLib` module, which is automatically instantiated when this module is used.)

```
SYMBOL ChkBeforeDef COMPUTE
   IF (CoordLess (COORDREF, GetDefPt (THIS.Key, COORDREF)),
   message (ERROR, CatStrInd ("identifier occurs before its definition: ",
                              THIS.Sym),
            0, COORDREF))
   DEPENDS_ON INCLUDING DefPtRangeSetFirst.GotDefPt;
END;
```

The role `ChkBeforeDef` is then associated to all grammar symbols that represent applied identifer occurreces which shall be checked.

4.4 Check for Unique Object Occurrences

This module associates a boolean property `NAMEUnique` to object keys. It has the value 1 if the object occurs only once in the SYMBOL context `NAMEUnique`. It has the value 0 if it occurs more than once in the `NAMEUnique` context; otherwise the property is not set. The final value of the property is obtained by the attribute `NAMEUnique.NAMEUnique`, e.g. used to issue a message indicating multiple occurrences. (The same task can be solved using the more general module, See Section 4.2 [OccCnt], page 47.)

The module is instantiated by

```
$/Prop/Unique.gnrc+instance=NAME +referto=KEY :inst
```

The range for these computations has to be specified by the role `NAMERangeUnique` (see Section 4.1 [Usage], page 43).

The multiply defined check for our running example, as explained for the `OccCnt` module (see Section 4.2 [OccCnt], page 47), can be also achieved by:

```
SYMBOL Program       INHERITS RangeUnique END;
SYMBOL MultDefChk    INHERITS Unique COMPUTE
   IF (NOT (THIS.Unique),
   message (ERROR, CatStrInd ("identifier is multiply defined: ", THIS.Sym),
```

```
                 0, COORDREF));
    END;
```

4.5 Determine First Object Occurrence

This module determines whether a `NAMEFirstOcc` occurrence of an object is the first one in left-to-right depth-first tree order. The result is obtained by the attribute `NAMEFirstOcc.IsNAMEFirstOcc` that has the value 1 if it is the first occurrence, 0 otherwise. The computations of the module use the property named `NAMEFirstOcc`. (The same task can be solved using the more general module `OccCnt`, See Section 4.2 [OccCnt], page 47.)

The range for these computations has to be specified by the role `NAMERangeFirstOcc` (see Section 4.1 [Usage], page 43).

The module is instantiated by

```
    $/Prop/FirstOcc.gnrc+instance=NAME +referto=KEY :inst
```

4.6 Map Objects to Integers

This module computes a mapping of object keys to numbers starting from 0 incrementing by 1. The mapping is associated as property named `NAMEObjNo` to object keys. The mapping is unique within the `NAMERangeObjCnt`.

The module can be used to just count the objects that occur in a range, to prepare for generating unique identifier names on output, or to map objects of a range to addresses that are incremented by 1.

The module is instantiated by

```
    $/Prop/ObjCnt.gnrc+instance=NAME +referto=KEY :inst
```

The role `NAMEObjCnt` has to be associated to grammar symbols such that all objects that should be considered have an occurrence in such a context. The attribute `NAMEObjCnt.NAMEObjNo` is the number the object is mapped to.

A range for these computations is specified by the role `NAMERangeObjCnt` (see Section 4.1 [Usage], page 43). The attribute `NAMERangeObjCnt.NAMETotalObjNo` is the total number of objects in that range. The ranges may be nested. The mapping starts anew from 0 for each range node. The mappings of inner ranges do not contribute to outer ranges.

If the `ObjNo` property is accessed by user's computations, they have to state `NAMERangeObjCnt.NAMETotalObjNo` as a precondition.

We demonstrate an application of this module by mapping the objects of out running example to unique numbers, in order to print unique names as they resulted from the consistent renaming task. For that purpose the grammar root can be chosen for the range role. The `ObjCnt` is simply attached to the `IdentOcc` role which represents any identifier occurrence in our example:

```
SYMBOL Program INHERITS RangeObjCnt COMPUTE
  printf ("the program references %d different objects\n",
          THIS.TotalObjNo);
END;
SYMBOL IdentOcc INHERITS ObjCnt COMPUTE
  printf ("object %s%d referenced in line %d\n",
          StringTable (THIS.Sym), THIS.ObjNo, LINE);
END;
```

4.7 Associate Kinds to Objects

Objects in an input text are often classified to belong to one of several kinds, e.g. variables, procedures or labels in programming languages. They may occur in different contexts which determine their kind, require that they belong to a certain kind, or select different computations depending on their kind. Such a classification is often the part of the type analysis task.

This module can be used for any unique classification of objects which is encoded by integral values.

If Objects may belong to more than one kind, or occurrences allow for objects of several kinds the module `KindSet` (see Section 4.8 [KindSet], page 53) should be used instead of this one.

The module is instantiated by

```
$/Prop/Kind.gnrc+instance=NAME +referto=KEY :inst
```

This module associates a property named `NAMEKind` of type `int` to objects. Two computational roles `NAMESetKind` and `NAMEGetKind` are provided.

In a context `NAMESetKind` the `NAMEKind` property of the object is set to the value of the attribute `NAMESetKind.NAMEKind`, which has to be provided by a user's computation. In case that different `Kind` values are stated for one object in some `NAMESetKind` contexts the property value 0 named `IntMultiple` is associated.

In a context `NAMEGetKind` the property is accessed and supplied by the attribute `NAMEGetKind.HasNAMEKind`. It can be used to check if it is the required kind, or if the kind is ambigously set (`IntMultiple`), or if the kind is not set at all (value -1 named (`IntNone`).

The roles `NAMESetKind` and `NAMEGetKind` may be associated to the same grammar symbol. That is necessary if kinds are determined by applications of objects rather than by definitions, or if a language does not distinguish between defining and applied occurrences.

The role that specifies the range for these computations is `NAMERangeKind` (see Section 4.1 [Usage], page 43). If the `NAMEKind` property is accessed in other user's computations, those have to state `NAMERangeKind.GotNAMEKind` as precondition.

4.8 Associate Sets of Kinds to Objects

Objects in an input text are often classified to belong to one or more of several kinds, e.g. variables, procedures or labels in programming languages. They may occur in different contexts which determine their kind, require that they belong to a certain kind, or select different computations depending on their kind. Such a classification is often the part of the type analysis task.

This module can be used for any classification of objects which is encoded by non negative integral values.

The module is instantiated by

```
$/Prop/KindSet.gnrc+instance=NAME +referto=KEY :inst
```

The module uses sets of kinds implemented by values of type `unsigned int` provided by the `IntSet` module (see Section 8.4 [IntSet], page 104). (That module is instantiated automatically with `referto` parameter `int`, and the `instance` parameter omitted.) Hence the largest code chosen for a kind value must be less than the number of bits of an `unsigned int` (16 or 32 implementation dependent).

This module associates a property named `NAMEKindSet` of type `IntSet` to objects. Three computational roles `NAMEAddKind`, `NAMEAddKindSet`, and `NAMEGetKindSet` are provided.

In a context `NAMEAddKind` the kind value of the attribute `NAMEAddKind.NAMEKind` is added to the set of kinds of the object. The attribute has to be provided by a user's computation.

Similarly in a context `NAMEAddKindSet` the `IntSet` value of the attribute `NAMEAddKindSet.NAMEKindSet` is united to the set of kinds of the object. The attribute has to be provided by a user's computation.

In a context `NAMEGetKindSet` the property is accessed and supplied by the attribute `NAMEGetKindSet.HasNAMEKindSet`. It can be used to compare the set of required kinds to the set of associated kinds using functions of the `IntSet` module.

The roles `AddKind` and `AddKindSet` must not be associated to the same grammar symbol. `GetKindSet` may be combined with one of them. That is necessary if kinds are determined by applications of objects rather than by definitions, or if a language does not distinguish between defining and applied occurrences.

The role that specifies the range for these computations is `NAMERangeKindSet` (see Section 4.1 [Usage], page 43). If the `NAMEKindSet` property is accessed in other user's computations, those have to state `NAMERangeKindSet.GotNAMEKind` as precondition.

This module also provides three operations that modify `NAMEKindSet` properties stored in the definition module:

`InsertNAMEKindSet(k,i)` inserts element `i` into the set stored for key `k`, and yields the new set as result.

`UnionNAMEKindSet(k,s)` adds the set `s` to the set stored for key `k`, stores the result and returns it.

`IntersectNAMEKindSet(k,s)` intersects the set `s` with set stored for key `k`, stores the result and returns it.

We demonstrate the use of this module in our running example. It shall be analysed if each variable occurs at least once on the lefthand side of an assignment and on the righthand side. Hence, we introduce the kinds `VarAssigned` and `VarUsed`. A variable can have any set of the values depending on its occurrences. The values are named by a `.head` specification:

```
#define   VarAssigned    1
#define   VarUsed        2
```

In our tree grammar the two occurrences of variables can be distinguished for the symbol `Variable` rather than for the symbol `UseIdent`. Hence we propagate the `Key` attribute of `UseIdent` upto `Variable` and apply the module roles there:

```
SYMBOL Variable: Key: DefTableKey;
RULE: Variable ::= UseIdent COMPUTE
  Variable.Key = UseIdent.Key;
END;

RULE: Statement::= Variable '=' Expression ';' COMPUTE
  Variable.Kind = VarAssigned;
END;

RULE: Expression ::= Variable COMPUTE
  Variable.Kind = VarUsed
END;

SYMBOL Variable INHERITS AddKind END;
SYMBOL Program  INHERITS RangeKindSet END;
```

In the context of a variable declaration the set of kinds is checked using functions of the `IntSet` module:

```
SYMBOL DefIdent INHERITS GetKindSet END;
RULE: ObjDecl ::= TypeDenoter DefIdent COMPUTE
  IF (NOT (InIS (VarAssigned, DefIdent.HasKindSet)),
  printf ("variable %s declared in line %d is never assigned\n",
           StringTable (DefIdent.Sym), LINE));

  IF (NOT (InIS (VarUsed, DefIdent.HasKindSet)),
  printf ("variable %s declared in line %d is never assigned\n",
           StringTable (DefIdent.Sym), LINE));
END;
```

4.9 Reflexive Relations Between Objects

This module introduces properties that relate object keys pairwise to each other, e.g. a type and its pointer type.

The module is instantiated by

```
$/Prop/Reflex.gnrc+instance=NAME :inst
```

It defines a pair of properties `NAMETo` and `NAMEFrom` that have values of type `DefTableKey` that relate keys pairwise to each other. When the relation is established between two keys `kf` and `kt` `GetNAMETo (kf, NoKey) == kt` and `GetNAMEFrom (kt, NoKey) == kf` hold.

The relation is established by a call `ReflexNAMETo (kf)` that yields a new key `kt`, or by `ReflexNAMEFrom (kt)` that yields a new key `kf`. Any further such call yields the same key as result.

Typical applications of such relations are found in type analysis tasks: Types can be represented by keys. Assume `intKey` represents the type `int`, then a call `ReflexPointerTo (intKey)` yields a key representing a type `pointer to int`. Using the `Reflex` functions guarantee that there is exactly one key representing the type `pointer to int`. Here the module is instantiated with the generic parametr `+instance=Pointer`. The same pattern can be applied for other unary type constructors.

4.10 Some Useful PDL Specifications

This module specifies a set of useful generic PDL patterns. If such patterns are associated to a property specification PDL generates additional access functions for that property.

For example the PDL property specification

```
Size: int [SetGet, SetDiff];
```

allows to set the `Size` property using the functions `SetGetSize` and `SetDiffSize` besides the basic access functions provided by PDL.

The module does not have generic parameters. It is used by writing

```
$/Prop/PropLib.fw
```

in a `.specs` file.

It provides the following PDL patterns:

`SetGet`: same as the basic `Set`, but the associated value is returned as result of the call.

`SetOnce`: has one value argument, the given value is only set if that property is not set; the associated value is returned as result of the call.

`KReset`: same as Reset, but the key is returned as result of the call, allowing nested calls for property association.

`Trans`: has only the key argument; for properties of type `DefTableKey`; the key at the end of the property chain is returned. The property chain must not be cyclic. E.g. if `GetTypeOf (a, NoKey) == b` and `GetTypeOf (b, NoKey) == c` and `GetTypeOf (c, NoKey) == NoKey`, then `TransTypeOf (a) == c`.

`SetDiff`: has two value arguments, if the property is not yet associated a value the first value argument is set; if the property has a value that differs from the first value argument, the second value argument is set.

The module also provides comparison functions `CoordLess` and `CoordLessEqual` for source coordinates.

5 Type Analysis

In programming languages types are used to classify entities the program operates on. A type describes a set of values and the operations that can be applied to them. Program constructs that denote entities are said to have a type which restricts the set of values the construct may evaluate to when executing the program. Program constructs that denote operations may restrict the types of the operands they are applied to. The meaning of an operation may be determined by the types of the operands. Type analysis determines the type of program constructs and of defined objects, it distinguishes overloaded operators using type information, and checks consistency with language rules on types. The result of the type analysis task is a precondition for the transformation of data objects and operations into a target language.

In languages other than programming languages we often find similar concepts that need to solve a type analysis task for implementing that language.

The section (See Section 5.1 [BasicType], page 60) describes a module that provides a set of computational roles for language construct that define and use typed objects, that denote types, define and use names for them.

The section (See Section 5.3 [TypeProp], page 66) gives examples how to use the `BasicType` module for languages having user defined types (e.g. `array` types).

The section (See Section 5.8 [Operator], page 75) describes a module that supports specification and use of overloaded operators. This module fits to the `BasicType` module, but it does not require to use it.

The module (See Section 5.9 [Defer], page 79) provides computations that relate keys to each other which have the same properties associated. It is used if different identifiers shall conceptually refer to one object. The `BasicType` is based on the `Defer` module.

The modules of this library are based on the following implementation strategy:

Types are represented by `DefTableKey`s. Hence, properties can be associated to a type to further characterize the type, as described in (See Section 5.3 [TypeProp], page 66). `NoKey` is used to represent the unknown type.

The specified language may allow to define type identifiers. Their keys are related to the keys representing their types.

The property `TypeOf` associates the type of an object to its key.

Attributes are associated to symbols describing the type of a program construct.

A complete executable specification of our running example for the type analysis task is available in `$/Type/Examples/Type.specs`.

5.1 Basic Type Analysis

This module implements a set of concepts that are basic for the type analysis task in many statically typed languages: Definition and use of typed objects, notation of user defined types, and definition and use of type identifiers.

The use of this module is demonstrated in

`Type Example`
: Running Example

This module is instantiated by

```
$/Type/Typing.gnrc +referto=|KEY| :inst
```

The `referto` parameter modifies the names of `Key` attributes, and hence, has to be the same as the `referto` parameter used for the module instance that supplied those attributes.

The module uses the following basic concepts for implementation of type analysis tasks:

Types are represented by `DefTableKeys`. Such a key is created for each program construct which denotes a particular type. The roles of this module associate the property `IsType` (set to 1) to such type keys in order to distinguish them from keys for other objects. Further properties may be associated to characterize the particular type.

`NoKey` represents the unknown Type.

The property `TypeOf` is associated to typed objects and states the type of the object.

Identifiers may be defined to denote types. They may be introduced by type definitions. In case of a type definition which simply renames the type given by a type identifier, it is assumed that both type identifiers refer to the same type.

This module uses an instance of the `Defer` module (see Section 5.9 [Defer], page 79) to relate keys that represent the same type. A type identifier key refers indirectly via the `Defer` relation to a type key, where the type properties are associated to. Any type property of a type denoted by a key `k` of a type identifier has to be accessed from the type key at the end of the `Defer` chain obtained by a call `TransDefer (k)`.

This module provides the following computational roles:

`RootType`: the root symbol.

`TypedDefinition`: a definition that defines one or several typed objects.

`TypedDefId`: a defining occurrence of an identifier for a typed object.

`TypedUseId`: an applied occurrence of an identifier for a typed object.

`ChkTypedUseId`: checks that the key represents a typed object.

`TypeDefDefId`: a defining occurrence of a type identifer.

`ChkTypeDefDefId`: checks that a type identifer is defined acyclic.

`TypeDefUseId`: an applied occurrence of a type identifer.

`ChkTypeDefUseId`: checks that the key represents a type.

`TypeDenotation`: a type denotation.

At least the roles `RootType`, `TypedDefinition`, `TypedDefId`, `TypedUseId`, `TypedUseId`, and `TypeDefUseId` are used. Those are sufficient for languages that have only predefined types. User defined types require the use of `TypeDenotation`. For definitions of names that denote types `TypeDefDefId` and `ChkTypeDefDefId` are necessary.

The roles are described as follows:

`TypedDefinition` is a context where one or several typed objects are defined in its subtree, using the role `TypedDefId`. They all have the same type which is to be computed as the attribute `THIS.Type`.

`TypedDefId` is a defining occurrence of an identifier for a typed object. The `TypeOf` property is associated to `THIS.KEYKey`. It is obtained from the attribute `INH.Type`. A default computation for `INH.Type` is provided that takes the type from `INCLUDING TypedDefinition.Type`. Further properties may be associated to the object along with its type. Then the postcondition of their computation should be `INH.GotProp` which is empty by default.

`TypedUseId` is an applied occurrence of an identifier for a typed object. The attribute `SYNT.Type` is the type associated to the object. The `Defer` chain is already walked down to the type key.

`ChkTypedUseId` checks that the key represents a typed object.

Types are denoted either by a used occurrence of a type identifier, role `TypeDefUseId`, or by some composite `TypeDenotation`. Either role provides a `Type` attribute representing the denoted type.

`TypeDefUseId` is an applied occurrence of a type identifier. The `Type` attribute is its key which is related to the type key by the `Defer` relation.

`ChkTypeDefUseId` is usually inherited along with `TypeDefUseId`. It issues an error message if the identifier does not denote a type.

`TypeDenotation` characterizes a context that denotes a user defined type, e.g. an array type. The attribute `SYNT.Type` is a new, unique type key. If the type rules of the language state that a thus denoted type is equivalent to any other type, suitable functions that check for type equivalence have to be used.

Any property that further describes the type has to be associated to `SYNT.Type` by computations that establish the postcondition `SYNT.GotType`. (Its default is the empty postcondition.) It guarantees that the properties can be accessed when object having that type is used (role `TypedUseId`).

`TypeDefDefId` is a defining occurrences of a type identifier. Its key is related by the `Defer` relation to the attribute `THIS.Type`. Its computation has to be provided.

`ChkTypeDefDefId` checks that its attribute `THIS.Type` refers to a type, and that the type does not refer to itself. It is usually inherited along with `TypeDefDefId`.

`RootType` has to be inherited to a symbol that contains all occurrences of any other role of this module. It ensures that all `TypeOf` and `Defer` properties are established when a `Type` attribute is computed. (The same holds for properties that are defined along with types.) That condition is provided by the condition attribute `SYNT.GotType`. Computations that access type properties, e.g. those which decompose composite types, have to be specified to depend on `INCLUDING RootType.GotType`.

5.2 Example for Basic Type Analysis

The use of the basic type module is demonstrated by introducing type analysis to our running example.

A complete executable specification of our running example for the type analysis task is available in `$/Type/Examples/Type.specs`.

For the moment we assume the there are only three predefined types `int`, `float`, and `bool`. The are represented by the keys `intType`, `realType`, and `boolType` as introduced in (see Section 3.6 [Predefined Identifiers], page 22).

We add a `.pdl` specification which states that those keys are types and which defines the types of the constants `true` and `false`. A `voidType` is introduced here although there is no predefined identifier to denote it. It enables us to express that in certain contexts type coercion to `voidType` is to be applied.

```
intType -> IsType = {1};
realType -> IsType = {1};
boolType -> IsType = {1};
```

```
voidType -> IsType = {1};

trueKey -> TypeOf = {boolType};
falseKey-> TypeOf = {boolType};

intKey -> Defer = {intType};
realKey -> Defer = {realType};
boolKey -> Defer = {boolType};
```

The last group relates the keys of the predefined type identifiers to the type keys. It is not recommended to use `intKey` etc. themselves as type keys. In that case it would not be possible to have different predefined type identifiers that could denote the same types.

The following association of module roles to grammar symbols in our `.lido` specification is obvious:

```
SYMBOL Program INHERITS RootType END;
SYMBOL DefIdent INHERITS TypedDefId END;
SYMBOL UseIdent INHERITS TypedUseId, ChkTypedUseId END;
SYMBOL TypeUseIdent INHERITS TypeDefUseId, ChkTypeDefUseId END;
```

Our language construct `ObjDecl` defines typed objects. It requires to specify the `Type` attribute to be taken from the `TypeDenoter`. Until now, a `TypeDenoter` can have only one form: `TypeUseIdent`. Its key is taken to represent the denoted type:

```
SYMBOL ObjDecl INHERITS TypedDefinition END;
RULE: ObjDecl ::= TypeDenoter DefIdent COMPUTE
  ObjDecl.Type = TypeDenoter.Type;
END;

RULE: TypeDenoter ::= TypeUseIdent COMPUTE
  TypeDenoter.Type = TypeUseIdent.Key;
END;

ATTR Type: DefTableKey;
```

The last line allows us to use `Type` attributes without specifying their type repeatedly.

The types of literal expressions is then specified by

```
RULE: Expression ::= IntNumber COMPUTE
  Expression.Type = intType;
END;
```

```
RULE: Expression ::= RealNumber COMPUTE
  Expression.Type = realType;
END;
```

Computations like the following propagate type information upwards through chain contexts:

```
RULE: Expression ::= Variable COMPUTE
  Expression.Type = Variable.Type;
END;

RULE: Variable ::= UseIdent COMPUTE
  Variable.Type = UseIdent.Type;
END;
```

Occurrences of `Expression` in certain contexts require that their type is compatible to a type imposed by the context. Hence, we introduce an inherited attribute `ReqType` and a check for compatibility. The two rules demonstrate how such type requirements are imposed:

```
SYMBOL Expression: ReqType: DefTableKey;
SYMBOL Expression COMPUTE
  IF (NOT (Compatible (THIS.ReqType, THIS.Type)),
  message (FATAL, "expression does not have the required type",
           0, COORDREF));
END;

RULE: Statement ::=  Expression ';' COMPUTE
  Expression.ReqType = voidKey;
END;

RULE: Statement ::= Variable '=' Expression ';' COMPUTE
  Expression.ReqType = Variable.Type;
END;
```

The function `Compatible` needs to be implemented according to the type rules of the language. (An example implementation is shown in `$/Type/Examples/Type.specs`).

We now introduce type definitions to our language. The concrete production is

```
Declaration:    'type' TypeDenoter TypeDefIdent ';'.
TypeDefIdent:   Ident.
```

From the view of language design such type definitions only make sense if there are other than the predefined types that can be given a name. In (see Section 5.3 [TypeProp], page 66) it is demonstrated how to introduce some common types.

The `TypeDefIdent` has the role of an defining type identifier occurrence that renames the type of the `TypeDenoter`:

```
SYMBOL TypeDefIdent INHERITS
       IdDefScope, IdentOcc,
       TypeDefDefId, ChkTypeDefDefId
END;

RULE: Declaration ::= 'type' TypeDenoter TypeDefIdent ';' COMPUTE
  TypeDefIdent.Type = TypeDenoter.Type;
END;
```

5.3 Properties of Types

In general it is necessary to associate properties to types that carry characterizing information which is either used for purposes of type analysis, e.g. to check compatibility of two types or to determine the element type of an array, or for translation purposes, e.g. the size of data objects in the target code. It is an important design issue in type analysis to identify the characterizing properties of types.

In this section four common type classes are used to demonstrate principles of type properties, and their use in type analysis. Solutions for other types may be derived from these by analogy. Here only properties are described that are used in type analysis itself, disregarding properties for the transformation task.

For our solution we use the module for basic type analysis (see Section 5.1 [BasicType], page 60) and some modules of the property library (see Chapter 4 [Prop], page 43).

`ArrayType`

    Array Types

`PointerType`

    Pointer Types

`FunctionType`

    Function Types

`ScopeTypes`
Types Having Scope Properties

Array types and pointer types are used to demonstrate how new types are introduced by declarations in a program, as opposed to predefined types, and that type properties refer to other types.

Function types are used to demonstrate type properties that are lists of types which are specified using the `LidoList` module.

Record types are used to demonstrate type properties that are scopes which are specified using modules of the `Name` library.

5.4 Array Types

In this section we use array types to demonstrate how new types are introduced by declarations in a program, as opposed to predefined types. Furthermore it is shown how type properties refer to other types. We assume that an array is described by the type and the number of its elements, like in C. The denotation of an array type shall introduce a new type that is diffierent from any other type, as in Pascal. For ease of our example we do not consider type rules that allow two different array types to be compatible.

The language of our running example is extended by `TypeDenoter`s for arrays and by indexed `Variable`s as specified by the concrete productions:

```
TypeDenoter:    ArrayType.
ArrayType:      TypeDenoter '[' IntNumber ']'.
Variable:       Variable '[' Expression ']'.
```

The form chosen for `TypeDenoter`s keeps the grammar simple and orthogonal, but leads to a bit unconventional notation for declaration of array variables and array type names:

```
type int[10]   Vector;
var  Vector    v;
var  float[5]  a;

v[6] = 1; a[3] = 2.5;
```

An array type is described here by two properties introduced by the `.pdl` specification

```
ElemType:       DefTableKey;
ElemNo:         int;
```

The rule that any type denoter introduces a new array type is reflected by using the module role `TypeDenotation` unchanged (see Section 5.1 [BasicType], page 60). It provides a computation for `ArrayType.Type` being a new, unique type key:

```
SYMBOL ArrayType INHERITS TypeDenotation END;

RULE: ArrayType ::= TypeDenoter '[' IntNumber ']' COMPUTE
  ArrayType.GotType =
    ORDER
      (ResetElemType (ArrayType.Type, TypeDenoter.Type),
       ResetElemNo (ArrayType.Type, IntNumber));
END;

RULE: TypeDenoter ::= ArrayType COMPUTE
  TypeDenoter.Type = ArrayType.Type;
END;
```

Association of the type properties establishes the postcondition `ArrayType.GotType`. The module roles ensure that the properties are set before accessed.

In the context of an indexed `Variable` the type is expected to be an array type, as checked in the second computation. The first computation accesses the element type. The call of `TransDefer` ensures that `Type` attributes in expression contexts are type keys rather than deferred keys. This is a convention that is introduced by the module roles, and is carried on here.

```
RULE: Variable ::= Variable '[' Expression ']' COMPUTE
  Variable[1].Type =
     TransDefer (GetElemType (Variable[2].Type), NoKey));

  IF (EQ (Variable[1].Type, NoKey),
  message (ERROR, "index applied to non array", 0, COORDREF));

  Expression.ReqType = intType;
END;
```

5.5 Pointer Types

In this section we introduce pointer types to the language of our running example. A type denoted `t !` shall be the type of objects that point to objects of type `t`. Types `t !` and `s !` shall be equivalent if `s` and `t` are equivalent, due to type definitions that simply rename a type. (This rule is similar to that of C, but different from that of Pascal where any ocurrence of a pointer type denoter introduces a new type.)

The concrete grammar of our running example is extended by the productions:

```
TypeDenoter:    PointerType.
PointerType:    TypeDenoter '!'.
Variable:       Variable '!'.
Variable:       Variable '&'.
```

Two new `Variable` notations are introduced: `v !` denotes the object which the value of the pointer variable `v` points to. `v &` yields the address of the variable `v`, it has the type pointer to `t` if `v` is a variable of type `t`.

Due to our equivalence rules all variables in the following example have the equivalent types:

```
type int ! IntPtr;
type IntPtr IP;
var  IntPtr i, IP j, int ! k;
...
i! = j!;
```

The property `PointsTo` is introduced to describe pointer types:

```
PointsTo: DefTableKey [KReset];
```

(The operation `KReset` is described below.)

The following computation introduces the notation of pointer types using the module role `TypeDenotation`. The computation establishes the postcondition as required for `TypeDenotations`:

```
SYMBOL PointerType INHERITS TypeDenotation END;
```

```
RULE: PointerType ::= TypeDenoter '!' COMPUTE
  PointerType.GotType =
    ResetPointsTo (PointerType.Type, TypeDenoter.Type);
END;

RULE: TypeDenoter ::= PointerType COMPUTE
   TypeDenoter.Type = PointerType.Type;
END;
```

If we do not take further means any two occurrences of a pointer type notation would denote different types. The equivalence rule described above has to be implemented in functions which compare types. Caution has to be taken to avoid non-termination of such functions if applied to recursively defined types. More details can be found in `$/Type/Examples/Type.specs`.

For the contents operation applied to a `Variable` we access the `PointsTo` relation that holds if the `Variable` really has a pointer type:

```
RULE: Variable ::= Variable '!' COMPUTE
  Variable[1].Type =
    TransDefer (GetPointsTo (Variable[2].Type, NoKey));

  IF (EQ (Variable[1].Type, NoKey),
  message (ERROR, "is not a pointer variable", 0, COORDREF));
END;
```

The call of `TransDefer` ensures that `Type` attributes in expression contexts are type keys rather than deferred keys. A convention that is introduced by the module roles, and carried on here.

The address operator implicitly creates a pointer type that `PointsTo` the type of the variable. Both properties `PointsTo` and `IsType` are set for that new type key.

```
RULE: Variable ::= Variable '&' COMPUTE
  Variable[1].Type =
     KResetPointsTo
       (KResetIsType (NewKey (), 1),
        Variable[2].Type);
END;
```

The `KReset` operations set the property and return the first argument. That mechanism is obtained from the module See Section 4.10 [PropLib], page 56 which is automatically instantiated when using the basic type module.

5.6 Function Types

Function types are an example for types that have properties which are lists of types, the types of the parameters. They are also used here to demonstrate an application of the `LidoList` module (see Section 8.1 [LidoList], page 97).

The type of a function is described by its result type and the list of the types of the parameters. Type analysis of a function call has to check the parameter types of the called function against the corresponding argument types, and has to yield the result type as the type of the call.

We extend the concrete grammar of our running example by productions for function declarations and calls:

```
Declaration:    FunctionDecl.
FunctionDecl:   'fun' DefIdent Function ';'.
Function:       FunctionHead Block.
FunctionHead:   '(' Parameters ')' TypeDenoter.
Parameters:     Parameter // ','.
Parameter:      TypeDenoter Ident.
Expression:     Expression '(' Arguments ')'.
Arguments:      Args.
Args:           .
Args:           Argument // ','.
Argument:       Expression.
```

In the context of a function declaration the list of parameter types is composed and associated as a property of the function type. In the context of a function call that property is accessed, the list is decomposed, and its elements - the formal parameter types - are compared with the types of the arguments.

In order to describe lists of parameter types we instantiate the `LidoList` module for the element type `DefTableKey` as described in (See Section 8.1 [LidoList], page 97):

```
$/Adt/LidoList.gnrc+instance=DefTableKey+referto=deftbl:inst
```

A function type is described by two properties, the result type and and the list of parameter types as introduced by the following PDL specifications:

```
ParamTypes:     DefTableKeyList; "DefTableKeyList.h"
ResultType:     DefTableKey;
```

where the `DefTableKeyList` type and the file defining it is obtained from the module instance.

Several module roles are to be combined in the contexts of function declarations: The `Function` is a `RangeScope` where the parameters are bound. The `FunctionDecl` is a `TypedDefinition` for the function identifier. The `FunctionHead` is a `TypeDenotation` describing the signature of the function. The `Parameter` is a `TypedDefinition`.

```
SYMBOL Function INHERITS RangeScope END;

SYMBOL FunctionDecl INHERITS TypedDefinition END;
RULE: FunctionDecl ::= 'fun' DefIdent Function ';' COMPUTE
  FunctionDecl.Type = Function.Type;
END;

SYMBOL FunctionHead INHERITS TypeDenotation END;
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter COMPUTE
  FunctionHead.GotType =
    ORDER
      (ResetResultType (FunctionHead.Type, TypeDenoter.Type),
       ResetParamTypes (FunctionHead.Type,
                        Parameters.DefTableKeyList));
END;
SYMBOL Parameter INHERITS TypedDefinition END;
```

Language rules that determine when the types of two functions are equal have to be implemented in the functions for type comparison. An example can be found in `$/Type/Examples/Type.specs`.

The parameter type list is composed using the list construction roles of the `LidoList` module:

```
SYMBOL Parameters INHERITS DefTableKeyListRoot END;
SYMBOL Parameter  INHERITS DefTableKeyListElem END;
RULE: Parameter ::= TypeDenoter DefIdent COMPUTE
  Parameter.DefTableKeyElem = TypeDenoter.Type;
END;
```

In the context of a call the type of the `Expression` is expected to be a function type. The parameter type list is decomposed using the list decomposition roles of the `LidoList` module, and checked for missing arguments:

```
SYMBOL Arguments INHERITS DefTableKeyDeListRoot END;
RULE:  Expression ::= Expression '(' Arguments ')' COMPUTE
  Arguments.DefTableKeyList =
     GetParamTypes (Expression[2].Type, NULLDefTableKeyList);
```

```
      Expression[1].Type =
         TransDefer (GetResultType (Expression[2].Type, NoKey));

      IF (EQ (Expression[1].Type, NoKey),
      message (ERROR, "call applied to non function", 0, COORDREF));

      Expression[2].ReqType = Expression[2].Type;

      IF (NE (Arguments.DefTableKeyListTail, NULLDefTableKeyList),
      message (ERROR, "arguments missing", 0, COORDREF));
    END;
```

The type of each formal parameter is obtained from the decomposed list and stated to be the required type of the actual argument:

```
    SYMBOL Argument INHERITS DefTableKeyDeListElem END;
    RULE:  Argument ::= Expression COMPUTE
      Expression.ReqType = Argument.DefTableKeyElem;

      IF (EQ (Argument.DefTableKeyElem, NoDefTableKey),
      message (ERROR, "too many arguments", 0, COORDREF));
 END;
```

To obey the naming requirements of the `LidoList` module we have to define the name `NoDefTableKey` for missing list elements in a `.head` specification:

```
    #define NoDefTableKey NoKey
```

5.7 Types Having Scope Properties

In this section we demonstrate types that have a property which is a scope, i.e. a set of object definitions. Examples for such types are records as in Pascal or C, or classes as in object-oriented languages. This task of type analysis is closely related to the name analysis task: The value of such a property is the scope of a certain range, e.g. the definitions of record components, or the members of a class. Variables of such a type may be used in component or member selections, again, a name analysis task.

In See Section 3.11 [Scope Properties], page 29 it is shown how scopes are used as properties of defined entities. Here we extend that concept towards typed objects using record types as an example.

A notation for record types is introduced into our language:

```
type record int a; bool b; end r;
var r rv, int i;
rv.a = 1;
i = rv.a;
```

Here, a record type `r` is defined, a variable `vr` of that type is declared, and component selections are used.

The concrete grammar is extended by productions for record type notations and for component selections:

```
TypeDenoter:    RecordType.
RecordType:     'record' ObjDecls 'end'.
Variable:       Variable '.' SelectIdent.
SelectIdent:    Ident.
```

We use an instance of the See Section 3.12 [AlgScopeProp], page 31 module. The name analysis role `RangeScopeProp` specifies the record type to be a range. Its scope of component definitions is associated to the `ScopeKey`. The `ScopeKey` is specified to be the type key created by the role `TypeDenotation`. The default for the `GotType` attribute need not be overridden, because the `AlgScopeProp` module takes care for setting the scope property before using it:

```
SYMBOL RecordType INHERITS TypeDenotation, RangeScopeProp
COMPUTE
  SYNT.ScopeKey = SYNT.Type;
END;

RULE: TypeDenoter ::= RecordType COMPUTE
  TypeDenoter.Type = RecordType.Type;
END;
```

The selection identifier is bound in a scope that is obtained from a scope property. Hence, it has the same computational roles as specified for the `QualIdent` of the scope operator `::` in (see Section 3.12 [AlgScopeProp], page 31), i.e. `IdUseScopeProp`, `ChkIdUse`, and `IdentOcc`. For type analysis it has additionally the roles for an applied identifier occurrence of a typed object.

```
SYMBOL SelectIdent INHERITS
     IdUseScopeProp, ChkIdUse, IdentOcc,
     TypedUseId, ChkTypedUseId
END;
```

```
RULE:  Variable ::= Variable '.' SelectIdent COMPUTE
  SelectIdent.Scope = GetScope (Variable[2].Type, NoEnv)
     <- INCLUDING Program.GotScopeProp;

  Variable[1].Type = SelectIdent.Type;

  IF (EQ (SelectIdent.Scope, NoEnv),
  message (ERROR, "selection applied to non record type",
           0, COORDREF));
END;
```

Here the scope property for the selection is obtained from the type of the selected variable. The precondition ensures that the scope properties have been set.

5.8 Operator Identification

Operator identification is a subtask of type analysis for any language that has overloaded operators, i.e. a source operator symbol like `+` may denote one of several target operations, e.g. integer addition or floating point addition or boolean disjunction. The distinction is made using the types of the operands. (The Ada-like case where the expected result type is also used is not supported here.)

The `Operator` module allows compact descriptions of the characteristics of each operator. It is required that types of operands and expressions are represented by `DefTableKey`s which are PDL known keys. The module also supports transformation of operators. Target operators are also represented by PDL known keys which may have properties associated that describe their translation.

This module replaces the `AdaptOil` module of previous library versions, and covers all its functionality. It uses the current version of the Oil tool which supports operators and types being represented by `DefTableKey`s.

The `Operator` module is instantiated with a `referto` parameter that gives the name of the file containing the operator descriptions

```
$/Type/Operator.gnrc +referto=(FILENAME):inst
```

The `FILENAME` has to be parenthesized. There may be at most one instantiation of this module.

The file contains a sequence of macro calls that describe source operators and target operators. The module provides computational roles for binary and unary operators (`BinTgtOpr` and `UnTgtOpr`) which perform operator identification. Their use is described below.

The macros that describe the operators have the following forms:

`SrcOpr (Token, ConSymbol, TreeSymbol, SrcKey)`
: A source operator is described. `Token` is a literal terminal that denotes the operator in the concrete grammar. The operator's precedence is specified by the concrete grammar nonterminal `ConSymbol`, that collects a set of operators having the same precedence. In the tree grammar the operator (and all others of the `ConSymbol` set) is represented by a `TreeSymbol`. `SrcKey` is the name of a `DefTableKey` which represents this operator uniquely. It is automatically introduced as a PDL known key.

`TgtOpr (SrcKey, TgtKey, Signature, KeyInit)`
: A target operator `TgtKey` is described. It is uniquely represented by the `DefTableKey` `TgtKey`, which is automatically introduced as a PDL known key. The operator is one of several overloaded meanings for the source operator `SrcKey`. Its signature is specified by `Signature`, using the notation of the Oil tool. The types occurring in the `Signature` are names of `DefTableKey`s. Properties may be associated to `TgtKey` to be used for further analysis and translation tasks. They can be initialized by `KeyInit` which is a possibly empty `PropertyValueList` in PDL notation. (If more than one property is initialized, the macro name `comma` has to be used as a separator instead of the `,` token.)

The following restrictions must be obeyed: Every `SrcKey` must occur only once in a `SrcOpr` description. Every `TgtKey` must occur only once in a `TgtOpr` description.

The computational role `BinTgtOpr` is to be inherited to a grammar symbol that represents a binary operator and that is mentioned as a `TreeSymbol` in a `SrcOpr` description. Two attributes named `LType` and `RType` have to be set to the type of the left and of the right operand respectively. Attributes `ResType`, `LTType`, and `RTType` are computed representing the result type of the identified target operator and its operand types. An attribute `TgtKey` is computed representing the target operator by the key specified in the `TgtOpr` description.

The same holds for the computational role `UnTgtOpr`, with the exception that only the `RType` attribute has to be set.

The computational role `ChkOpr` provides a check whether a valid target operator is identified for the given types, and issues a message otherwise. It can be inherited to binary or unary operator symbols.

The use of this module is demonstrated by adding expressions with binary and unary operators to the language of our running example. In the concrete grammar an expression hierarchy defines precedences and associativity of operator sets:

```
Expression:     Expression AddOpr Factor / Factor.
Factor:         Factor MulOpr Operand / Operand.
Operand:        MonOpr Operand.
Operand:        '(' Expression ')'.
```

Here `UnOpr`s have highest precedence and `AddOpr` have lowest. The operators of the three sets are not specified here. They are introduced by `SrcOpr` descriptions contained in the file the module is instantiated with, e.g.

```
SrcOpr ('+', AddOpr, BinOpr, AddKey)
SrcOpr ('*', MulOpr, BinOpr, MulKey)
SrcOpr ('-', MonOpr, UnOpr,  NegKey)
```

The module adds a production for every source operator to the concrete grammar, e.g.

```
AddOpr: '+'.
```

It states that `AddOpr` and `MulOpr` are to be represented by a `BinOpr` node in the tree, and it adds an attribute computations for each operator context to the tree, e.g.

```
RULE: BinOpr ::= '+' COMPUTE BinOpr.OprSym = AddKey; END;
```

The following target operator descriptions (also contained in the file the module is instantiated with) overload those operators with operations on integral numbers and boolean values:

```
TgtOpr (AddKey, iAddKey, (intKey,intKey):intKey,    TgtStr={"+"})
TgtOpr (AddKey, bOrKey,  (boolKey,boolKey):boolKey, TgtStr={"||"})
TgtOpr (MulKey, iMulKey, (intKey,intKey):intKey,    TgtStr={"*"})
TgtOpr (MulKey, bAndKey, (boolKey,boolKey):boolKey, TgtStr={"&&"})
TgtOpr (NegKey, iNegKey, (intKey):intKey,           TgtStr={"-"})
TgtOpr (NegKey, bNotKey, (boolKey):boolKey,         TgtStr={"!"})
```

From these descriptions the module generates the necessary .oil specifications, e.g

```
INDICATION AddKey: iAddKey;
OPER iAddKey (intKey,intKey):intKey;
```

and the necessary .pdl specifications, e.g

```
iAddKey -> TgtStr={"+"};
```

The properties that are initialized in the target operator description, here `TgtStr` have to be defined in the user's specification.

We then can use the computational roles `BinTgtOpr`, `UnTgtOpr`, and `ChkOpr` to solve the overloading resolution task. We only have to assign the types of the operands to attributes of the operator symbol, and use the types of the signature of the identified target operator correspondingly:

```
SYMBOL Expression: Type, ReqType: DefTableKey;

SYMBOL BinOpr INHERITS BinTgtOpr, ChkOpr END;
RULE: Expression ::= Expression BinOpr Expression COMPUTE
  BinOpr.LType = Expression[2].Type;
  BinOpr.RType = Expression[3].Type;
  Expression[1].Type = BinOpr.ResType;
  Expression[2].ReqType = BinOpr.LTType;
  Expression[3].ReqType = BinOpr.RTType;
END;

SYMBOL UnOpr INHERITS UnTgtOpr, ChkOpr END;
RULE: Expression ::= UnOpr Expression COMPUTE
  UnOpr.RType = Expression[2].Type;
  Expression[1].Type = UnOpr.ResType;
  Expression[2].ReqType = UnOpr.RTType;
END;
```

The operator descriptions of the target operators in our example associate a property `TgtStr` to be used for a translation to C:

```
RULE: Expression ::= Expression BinOpr Expression COMPUTE
  Expression[1].CCode =
     PTGBinOpr (Expression[2].CCode,
                GetTgtStr (BinOpr.TgtKey, "??"),
                Expression[3].CCode);
END;

RULE: Expression ::= UnOpr Expression COMPUTE
  Expression[1].CCode =
     PTGUnOpr (GetTgtStr (UnOpr.TgtKey, "??"),
               Expression[2].CCode);
END;
```

To complete the above example we need a .pdl specification of the `TgtStr` property

```
TgtStr:  CharPtr; "Strings.h"
```

and suitable `.ptg` patterns for the translation of expressions:

```
BinOpr: "(" $1 ")" $2 string "(" $3 ")"
UnOpr:  $1 string "(" $2 ")"
```

5.9 Deferred Property Association

This module implements the technique of deferred property association: Many languages have constructs that define an identifier to denote the same object as another, different identifier does. Properties accessed or set via the one key should yield the same results or effects as if the other key was used. Typical examples for such constructs are type definitions or constant definitions.

The module is instantiated by

```
$/Type/Defer.gnrc +referto=KEY :inst
```

The `referto` parameter modifies the names of `Key` attributes, and hence, has to be the same as the `referto` parameter used for the module instance that supplied those attributes.

The roles of this module relate keys to each other which represent the same object. That relation has to be acyclic. The properties are associated to the keys at the ends of those relation chains. A function is provided that walks down the chain when accessing a property from any of the related keys.

This technique also decouples the computations which establish the equivalence between key from those which associate properties to keys. It avoids cyclic dependencies between computations in cases where properties of objects may be defined recursively, e.g. recursively defined types.

The property `Defer` implements the relation between keys described here. It should not be set otherwise than by using the `SetDeferId` role of this module.

Setting a property to a key that may be an end of a `Defer` chain should occur in the context of the `SetDeferProp` role.

If properties are accessed for a key `k` that may be on a `Defer` chain, the result of the call `TransDefer (k)` has to be used instead of the the key `k` itself, e.g. `GetKind (TransDefer (k), NoKind)`.

This module uses the `PropLib` module (See Section 4.10 [PropLib], page 56) to obtain the `TransDefer` function.

The module provides the following computational roles:

`SetDeferId` is a role for a defining occurrence of an identifier. It establishes the `Defer` relation from `SetDeferId.|KEY|Key` to point to `SetDeferId.DeferredKey`. A lower or upper computation for `THIS.DeferredKey` has to be provided. An attempt to complete a `Defer` cycle is not executed.

`ChkSetDeferId` is a role that may be inherited to any identifier occurrence. It checks whether an attempt was made to complete a `Defer` cycle involving this key. The role should be inherited together with `SetDeferId`, if `Defer` cycles are not otherwise excluded.

`SetDeferProp` is a role that characterizes a context where properties may be set to a key at the end of a `Defer` chain. Computations that associate the properties have to establish the postcondition represented by the `VOID` attribute `SYNT.GotDeferProp`. The role provides a default computation for `SYNT.GotDeferProp` that states the empty postcondition.

6 Input Processing

This library contains modules that support insertion of files into the currently processed input file. The name of the file to be inserted is computed from information of the current input file. Such computations have to be executed immediately while the input is being read (bottom-up in terms of tree construction). This library also contains a module that supports command line arguments of the generated processor which specify directories where files to be included are searched.

`Include` is the central module to solve the file inclusion task. It is based on the modules `CoordMap` and `GlaCorr` which are automatically instatiated when `Include` is used. (`CoordMap` and `GlaCorr` are not intended to be used directly in specifications.) The module `CmdLineIncl` can be used additionally to get the command line feature.

The module `CurrTok` has to be used if the function provided by the `Include` module is issued from computations specified in the concrete grammar rather then from `.lido` computations.

6.1 Insert a File into the Input Stream

This module supports generating processors that include the contents of a file into the currently read input stream. The effect is the same as that known from the preprocessor `cpp`. There file inclusion is initiated by a `cpp` command like `#include "myfile"`, here it is initiated by a computation of the language processor. It usually takes the file name from the text of a token found in the input stream.

Such facilities are used in some systems for functional or logical languages, where commands like

```
use ["myfile"]
```

cause the inclusion of the given file at the place of the use command.

The file is included at the current position of the input stream when that computation is executed. That computation can either be specified within the concrete grammar to be initiated by the parser. Or it can be specified in a `.lido` specification. In the latter case the computation has to be executed while input is being read (and the tree is constructed bottom-up) to achieve the desired effect.

This module is instantiated without generic parameters by

```
$/Input/Include.gnrc:inst
```

The module provides the following function:

`int NewInput (char *name)`
: It switches input reading to the file named `name`. When its end is encountered input reading is switched back to the next not yet processed token in the file that was the current when `NewInput` was called. The call of `NewInput` yields 1 if the file could be opened; otherwise it yields 0.

This facility is demonstrated by inserting a construct into our running example that allows to comprise sets of declarations on separate files and include the in the program text. Such declaration files may be used to specify common interfaces.

The file inclusion construct is introduced by the concrete productions

```
Declaration:   FileInclusion ';'.
FileInclusion: 'include' FileName.
```

We add a scanner specification for the terminal symbol `FileName`:

```
FileName: C_STRING_LIT
```

Hence, an input program may contain file inclusion commands like `include "myfile";` wherever a `Declaration` is allowed.

The following `.lido` specification attaches the call of `NewInput` to the `FileInclusion` context:

```
ATTR InpFileEx: int;
RULE: FileInclusion ::= 'include' FileName COMPUTE
      .InpFileEx = NewInput (StringTable (FileName)) BOTTOMUP;

      IF (NOT (.InpFileEx),
      message (ERROR, CatStrInd ("can not open file ", FileName),
               0, COORDREF));
END;
```

The file name is specified to be the text of the `FileName` token. The computation has to executed while the input is read and the tree is built bottom-up. Hence it is specified `BOTTOMUP`.

Note: The above computation is executed when the `FileInclusion` tree node is built. Then the last token read from the current input stream is the ending ; of the `Declaration`, due to the lookahead of the parser. Hence, the contents of the included file is being processed as if it stood immediately following that ; of the `Declaration`. Due to this reason it was essential to use two productions in the concrete grammar as stated above. A single production like

```
Declaration:  'include' FileName ';'.
```

instead would delay insertion until the token following that ; has been read.

6.2 Accessing the Current Token

This module provides a function `GetCurrTok` that can be used to access the string of the current token read from input. The function is used for example to access the name of a file to be included if the switching computation is issued by the parser as specified in the concrete grammar.

This module is instantiated without generic parameters by

```
$/Input/CurrTok.gnrc:inst
```

The signature of the function is

`char *GetCurrTok (void)`

  It accesses the string of the current token, stores it in memory, and yields a pointer to the stored string as result.

In our running example we could achieve the same effect of file inclusion as described in (see Section 6.1 [Include], page 81), by specifying a computation in the concrete grammar rather than in a `.lido` specification:

```
Declaration: 'include'
             &'ChkFileOpnd(NewInput(GetCurrTok()), GetCurrTok());'
             FileName ';'.
```

The token of `FileName` that immediatly follows the computation is accessed. The call of `ChkFileOpnd` is assumed to check for success of the `NewInput` call.

6.3 Command Line Arguments for Included Files

Using this module introduces processing of a command line option into the generated processor: If the processor is called

```
processor.exe -Idirname filename
```

then files to be included are searched in the current directory and in `dirname`. This module provides specifications for the Eli's CLP tool to achieve the effect.

This module is instantiated without generic parameters by

```
$/Input/CmdLineIncl.gnrc:inst
```

The module provides a function that can be used together with calls of the `NewInput` function:

`char * FindFile (char *name)`

It searches the file named `name` in the current directory and in the directories given by `-I` command line arguments.

In order to use this facility the `FindFile` function has to be applied to the argument of the function `NewInput`. Hence, the examples of the two previous sections had to be rewritten as

```
ATTR InpFileEx: int;
RULE: FileInclusion ::= 'include' FileName COMPUTE
  .InpFileEx = NewInput (FindFile (StringTable (FileName))) BOTTOMUP;
```

```
    IF (NOT (.InpFileEx),
    message (ERROR, CatStrInd ("can not open file ", FileName),
             0, COORDREF));
  END;
```

for the case of the `.lido` specification of input switching as shown in (see Section 6.1 [Include], page 81), or

```
  Declaration: 'include' &'NewInput(FindFile (GetCurrTok()));' FileName ';'.
```

for the case of the concrete grammar specification of input switching as shown in (see Section 6.2 [CurrTok], page 83).

7 Generating Output

This library contains the following modules:

`LeafPtg` PTG Output for Leaf Nodes

`Indent` Indentation for PTG Output

`OutStr` Output String Conversion

`PrettyPrint`
PrettyPrinting for PTG Output

`BlockPrint`
Typesetting for Block-Sructured Output

7.1 PTG Output for Leaf Nodes

The module `LeafPtg` provides some standard translations of terminal input strings into output strings. They may be attached to nonterminal symbols having an `int` valued attribute `Sym` which represents a string, an identifier or a number. That symbol usually derives to terminal symbol which is used to compute the value of the `Sym` attribute.

The module does not have generic parameters. It is used by writing

```
$/Output/LeafPtg.fw
```

in a `.specs` file.

The computations of this module yield an attribute `Ptg` of type `PTGNode` representing the desired output string. It may used to compose more complex output structures.

Note: The computations of this module may not be inherited directly to terminals, as it was possible with older versions.

The required computation of the `Sym` attribute can be specified by (if not already done for other purposes):

```
ATTR Sym: int;
```

```
SYMBOL T COMPUTE SYNT.Sym = TERM; END;
```

There are the following symbol roles for different output representations:

```
SYMBOL T INHERITS  IdPtg  END;
```

`T.Sym` must refer to an identifier or a string then `T.Ptg` represents the same identifier or character sequence of the string in the output.

```
SYMBOL T INHERITS  CStringPtg  END;
```

`T.Sym` must refer to a string, then `T.Ptg` represents its value as a C string literal.

```
SYMBOL T INHERITS  PStringPtg  END;
```

`T.Sym` must refer to a string, then `T.Ptg` represents its value as a Pascal string literal.

```
SYMBOL T INHERITS  NumPtg  END;
```

`T.Sym` must represent an integral number, then `T.Ptg` represents that number.

The module introduces PTG specifications for patterns named `AsIs, CString, PString, Numb`. These names must not be specified in any other PTG specification.

On instantiation the `instance` and the `referto` parameter should not be used.

7.2 Indentation

The module `Indent` supplies a C module that implements some functions helpful for indenting text produced by PTG functions. The function names `IndentIncr`, `IndentDecr`, `IndentNewLine` can be inserted into the user's PTG specification, like

```
Block: "{" [IndentIncr] $ [IndentDecr] [IndentNewLine] "}"
Stmt:  [IndentNewLine] $ ";"
```

Those PTG specifications should not contain strings with the new line character, but should have the [IndentNewLine] call instead.

Use the function IndentNewLine to put a linefeed into the output and indent the next line. IndentIncr increments and IndentDecr decrements the indentation level. The width of a single indentation step may be set to n spaces by the call IndentSetStep (n) executed prior to initiating output (e.g. by PTGOut).

The module does not have generic parameters. It is used by writing

```
$/Output/Indent.fw
```

in a .specs file.

7.3 Output String Conversion

This module provides a set of functions that transform character and character string values into C or Pascal literals.

The module does not have generic parameters. It is used by writing

```
$/Output/OutStr.fw
```

in a .reqmod (or a .specs) file.

The module exports the following functions:

void C_outstr (FILE *fs, char *s)
: Translates s into a C string literal and outputs it on file fs.

void C_outchar (FILE *fs, char *s)
: Translates s into a C character literal and outputs it on file fs.

void P_outstr(FILE *fs, char *s)
: Translates s into a Pascal string literal and outputs it on file fs.

void outstr (FILE *fs, char *s)
: Outputs s without translation on file fs.

7.4 Pretty Printing

The module 'PrettyPrint' supplies C functions that can be inserted in PTG Patterns to handle proper wordwrapping. It is the aim of this module to break the current line at the last possible position that preceeds the maximum line width. Furthermore, regions of text can be indented.

Functions exist to mark linebreaks and the beginning and end of an indentation region. These functions can be included into PTG Pattern definitions.

`PP_BreakLine`
: Specifies, that a linebreak can be inserted at this point. A line will only be broken at these points.

`PP_Indent`
`PP_Exdent`
: `PP_Indent` marks the beginning of an indentation region. Linefeeds following this function call will not only begin a new line but also indent the next line by an indentation step, those width can be adjusted with a function discussed later. Indentation-regions can be nested and are terminated by a call to the `PP_Exdent`-pattern function.

`PP_Newline`
: Forces a linefeed thereby inserting the newline sequence. The next line will be indented properly. The newline character '"\n"' in a PTG Pattern is a shortcut for a call to this output function.

Examples

The following PTG Patterns can be used to yield different styles of indenting for blocks. Here { and } are the symbols that denote the beginning and end of a block, $ is the insertion point for the indented block. The first example sets those symbols in a new line at the indentation level of the outer block:

```
Block:  "\n{" [PP_Indent]
            "\n" $ [PP_Exdent]
        "\n}"
Stmt:   [PP_BreakLine] $
```

The next example also specifies an indentation region. Here, the opening brace is set as last token in the outside block, separated with whitespace instead of a newline:

```
Block:  " " [PP_BreakLine] "{" [PP_Indent]
           "\n" $ [PP_Exdent]
        "\n}"
```

The third example uses the indentation style commonly known as the GNU-Indentation-style. Here, the braces are set on a new line, indented two positions. The indented region then follows indented four positions. To use this, set the indentation width to two by one of the function calls discussed later. Then use the following pattern:

```
Block:  [PP_Indent] "\n{" [PP_Indent]
            "\n" $ [PP_Exdent]
        "\n}" [PP_Exdent]
```

Additional functions

Additional functions exist to influence the behaviour of the module.

`PP_SetLineWidth(int width)`
: Sets the linewidth to the specified value. The default is 80.

`PP_SetSoftBreakShortcut(char)`
: Assings a character that should behave like a call to `PP_LineBreak`. A good choice for this would be the tab character. The default is set to the null character what disables substitution.

`PP_SetIndentationWidth(int width)`
: Sets the amount to indent in one indentation level. Indentation is normally done by spaces. If a negative value is used, a tab character will be used for one indentation step (counting as 8 character positions).

`PP_SetEndline(char *endline)`
: Assigns the given string to be used as end-of-line sequence. Default for this is '`"\n"`'. Another good choice would be '`"\r\n"`'.

All these functions need to be called prior to start the output with one of the following functions. These replace the PTG generated ones, if PrettyPrinting should be used.

`PP_OutFPtr(FILE *f, PTGNode n)`
: Outputs the given '`PTGNode`' to the given file that must have been opened for output.

`PP_OutFile(char *filename, PTGNode n)`
: Outputs the given '`PTGNode`' to the named file.

`PP_Out(PTGNode n)`
: Outputs the given '`PTGNode`' to the standard output.

Usage of Module

To use the pretty printing module, simply include it's name in one of the `.specs` files:

```
$/Output/PrettyPrint.fw
```

Restrictions

In two cases it is possible that an output line exeeds the given maximal length:

A sequence of characters longer than the specified linewidth is output without intermediate call to `PP_LineBreak`.

A PTG Pattern contains tab characters that will be counted to have a width of 1 which of course is not always true.

Additional information about this module and it's implementation can be obtained by the derivation

```
$elipkg/Output/PrettyPrint.fw :fwTexinfo :display
```

7.5 Typesetting for Block Structured Output

The module '`BlockPrint`' supplies C functions that can be inserted in PTG Patterns to handle proper block formatting capabilities. It is the aim of this module to print all the text between the block marks into one line. If that does not succeed, all embedded linebreaks of the block are converted into newlines. Additionally, blocks can be nested and blocking can be combined with indentation.

Functions exist to mark linebreaks and the beginning and end of a block. These functions can be included into PTG Pattern definitions.

`BP_BreakLine`

Specifies, that a linebreak can be inserted at this point. A line will only be broken at these points.

`BP_BeginBlock`

`BP_EndBlock`

Marks the beginning and end of a Block. If the text until the call to `BP_BlockEnd` has room in the current line, all linebreaks will be discarded. Else, all embedded linebreaks will be converted into newlines.

`BP_BeginBlockI`

`BP_EndBlockI`

Same as the above. The block created by this pair of functions will aditionally be indented by one indentation step.

`BP_Newline`

Forces a linefeed thereby inserting the newline sequence. Note that with the presence of this pattern function, the enclosing block is automatically tagged as 'too long' and all the remaining Linebreaks in the current block are also converted to newlines. The next line will be indented properly. The newline character '`"\n"`' in a PTG Pattern is a shortcut for a call to this output function.

Examples

The following PTG Patterns can be used to print nested C scopes with intermediate function calls. The statements in one block will be indented properly and always be separated by newlines. The arguments of a function call will be set into one line, if there is enough room. If not, newlines will be inserted between the arguments.

```
FCall:          "\n" $ string "(" [BP_BeginBlockI] $ [BP_EndBlockI] ");"
Arg:            $ { "," [BP_BreakLine] } $
Block:          "\n{" [BP_BeginBlockI]
                    $ [BP_EndBlockI]
                "\n}"
```

Additional functions

`BP_SetLineWidth(int width)`

Sets the linewidth to the specified value. The default is 80.

`BP_SetSoftBreakShortcut(char)`

Assings a character that should behave like a call to `BP_LineBreak`. A good choice for this would be the tab character. The default is set to the null character what disables substitution.

`BP_SetIndentationWidth(int width)`

Sets the amount to indent in one indentation level. Indentation is normally done by spaces. If a negative value is used, a tab character will be used for one indentation step (counting as 8 character positions).

`BP_SetEndline(char *endline)`

Assigns the given string to be used as end-of-line sequence. Default for this is `"\n"`. Another good choice would be `"\r\n"`.

All these functions need to be called prior to start output with one of the following functions. These replace the PTG generated ones, if Block Printing should be used.

`BP_OutFPtr(FILE *f, PTGNode n)`

Output the given '`PTGNode`' to the given file that must have been opened for output.

`BP_OutFile(char *filename, PTGNode n)`

Output the given PTGNode to the named file.

`BP_Out(PTGNode n)`

Output the given PTGNode to the standard output.

Usage of Module

To use the block printing module, simply include it's name in one of the `.specs` files:

```
$/Output/BlockPrint.fw
```

Restrictions

In two cases it is possible that an output line exeeds the given maximal length:

A sequence of characters longer than the specified linewidth is output without intermediate call to `BP_BreakLine`.

A PTG Pattern contains tab characters that will be counted to have a width of 1 which of course is not always true.

Additional information about this module and it's implementation can be obtained by the derivation

```
$elipkg/Output/BlockPrint.fw :fwTexinfo :display
```

8 Abstract Data Types

This library provides implementations of abstract data types for linear lists, stacks, sets, and mappings by C modules. The functions exported by these modules can be used in `.lido` specifications or in C modules supplied by users.

This library contains the following modules:

`LidoList`	Lists in LIDO Specifications
`List`	Linear Lists of Any Type
`BitSet`	Bit Sets of Arbitrary Length
`IntSet`	Bit Sets of Integer Size
`Stack`	Stacks of Any Type
`Map`	Mapping of Integers to Any Type
`DynSpace`	Dynamic Storage Allocation

The module `LidoList` supports construction and access of linear lists in `.lido` specifications. The module `DynSpace` supports efficient dynamic storage allocation. It provides a simplified and restricted interface to the `obstack` module which is recommended to be used for dynamic allocation of arbitrary sized storage entities. The other modules are implementations of abstract data types.

The functions exported by each of these modules may be used in `.lido` computations as well as in C modules. In the latter case the interface file of the module has to be included into the C module.

8.1 Lists in LIDO Specifications

Several language implementation tasks are solved using linear lists constructed of data elements which are computed at certain tree nodes, e.g. a the list of parameter types that is part of the type of a function. Such lists reflect a left-to-right depth-first order of the corresponding tree nodes. Similarly such lists are decomposed in other contexts, e.g. to check the types of the arguments of a function call. This module provides `.lido` specifications for such patterns of linear list usage. The module uses functions of the linear list module `List`. Any assignable C type may be chosen as type of the list elements.

This module is instantiated by

```
$/Adt/LidoList.gnrc+instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the element type and `HDR.h` is a file that defines the element type, e.g.

```
$/Adt/LidoList.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/LidoList.gnrc+instance=int :inst
```

The module provides two pairs of computational roles, `TYPEListRoot` and `TYPEListElem` for list construction, and `TYPEDeListRoot` and `TYPEDeListElem` for list decomposition.

For construction of a list `TYPEListRoot` is associated to a grammar symbol that contains all occurrences of `TYPEListElem` for its list elements in its subtree. A user's computation has to provide the list element values of type `TYPE` by attributes `TYPEListElem.TYPEElem`. The resulting list of type `TYPEList` is obtained by the attribute `TYPEListRoot.TYPEList`.

For decomposition of a list `TYPEDeListRoot` is associated to a grammar symbol that contains in its subtree all occurrences of `TYPEDeListElem` for grammar symbols that get a list element value each. The list element values are obtained by attributes `TYPEDeListElem.TYPEElem`. If the list is shorter than the number of `TYPEDeListElem` occurrences the remaining attributes have the value `NoTYPE`, which has to be defined suitably in a user's specification, e.g. in `HDR.h`. `TYPEDeListRoot.TYPEListTail` is the list of remaining elements not associated to `TYPEDeListElem` occurrences, if any.

Both `TYPEListRoot` and `TYPEDeListRoot` may be recursively nested without affecting each other. Neither `TYPEListRoot` or `TYPEDeListRoot` may be associated to grammar symbols that directly produce no symbol or only terminal symbols.

An example for the use of this module in type analysis is given in (see Section 5.6 [FunctionType], page 71): In the context of a function declaration the list of parameter types is composed and associated as a property of the function type. In the context of a function call that property is accessed, the list is decomposed, and its elements - the formal parameter types - are compared with the types of the arguments.

8.2 Linear Lists of Any Type

This module implements linear lists whose elements are of an arbitrary type that is specified by a generic instantiation parameter. Any assignable type can be chosen.

Storage for lists is allocated when needed. The module implementation uses efficient dynamic storage allocation of the `obstack` module. The module does not implement automatic garbage collection. Storage used by one instance of this module can be deallocated completely.

One subset of the functions provided by this module is strictly functional, i.e. list values are not modified. Another subset of functions modifies existing list values, e.g. inserts elements into a list.

The module is instantiated by

```
$/Adt/List.gnrc +instance=TYPE +referto=HDR :inst
```

where `TYPE` is the name of the element type and `HDR.h` is a file that defines the element type, e.g.

```
$/Adt/List.gnrc+instance=DefTableKey +referto=deftbl :inst
```

If the element type is predefined in C the `referto` parameter is omitted, e.g.

```
$/Adt/List.gnrc+instance=int :inst
```

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `TYPEList.h` is imported there.

A module `PtrList` is available. It produces list implementations with the same interface as the `List` module does. `PtrList` ist only applicable if the element type `TYPE` is a pointer type. An instantiation of `PtrList` implements the lists by lists of `VoidPtr` (`void*`) using an instantiation of the `List` module. Hence, several instances of `PtrList` use one single implementation. The created interface files `TYPEList.h` provide macros for casting the element type to and from `VoidPtr`. The module `PtrList` is instantiated in the same way as the `List` module is:

```
$/Adt/PtrList.gnrc +instance=TYPE +referto=HDR :inst
```

The modules export the following type names and macros:

`TYPEList` A pointer type representing lists.

`TYPEListPtr`

A pointer type pointing to TYPEList objects.

`TYPEMapFct`

A function type for mapping: `TYPE -> TYPE`.

`TYPECmpFctType`

A function type for comparison: `TYPE, TYPE -> int` Function of this type have to yield 0 if the two values are equal, 1 if the left argument is greater than the right, -1 if the left argument is less than the right.

`TYPESumFct`

A function type for combining elements: `TYPE, TYPE -> TYPE`

`NULLTYPEList`

Denotes the empty list.

`NULLTYPEList ()`

Denotes the empty list.

The following list processing functions are supplied by the module:

`void FinlTYPEList (void)`

Deallocates all TYPELists.

`TYPEList ConsTYPEList (TYPE e, TYPEList l)`

Constructs a TYPEList of an element `e` and a given tail `l`. `e` is the first element of the list.

`TYPE HeadTYPEList (TYPEList l)`

Returns the first element of the list `l`. The list `l` must not be empty.

`TYPEList TailTYPEList (TYPEList l)`

Returns the tail of the list `l`. If `l` is empty, an empty list is returned.

`int LengthTYPEList (TYPEList l)`

Returns the number of elements in the list `l`.

`TYPE IthElemTYPEList (TYPEList l, int i);`

Returns the `i`-th element of the List `l`. The head of `l` is referred to as 1. If the value of i is greater than the length of the list, an error is reported and the program exits.

`TYPEList CopyTYPEList (TYPEList l, TYPEMapFct cp)`

Copies the list `l`. Elements are copied by calls of `cp`.

`TYPEList AppTYPEList (TYPEList l1, TYPEList l2)`

Concatenates two lists `l1` and `l2`. The resulting list contains `l2` at the end of a copy of list `l1`.

`TYPEList AppElTYPEList (TYPEList l, TYPE e)`

Appends an element `e` to the list `l`. The list `l` is not copied, it is modified as a side-effect of this function.

`void InsertAfterTYPEList (TYPEList l, TYPE e)`

This function requires a non-empty list `l`. The element `e` is inserted just after the first element of `l`. The list `l` is modified as a side-effect of this function.

`TYPEList OrderedInsertTYPEList (TYPEList l, TYPE e, TYPECmpFctType fcmp)`

Inserts the element `e` into the list `l` maintaining `l` in ascending order with respect to the compare `fcmp`. The list `l` is modified as a side-effect of this function.

`TYPEListPtr RefEndConsTYPEList (TYPEListPtr addr, TYPE e);`

Appends an element `e` to the end of a list given by its address `addr`. The address where the next element may be appended is returned. The list is modified as a side-effect of this function.

`TYPEListPtr RefEndAppTYPEList (TYPEListPtr addr, TYPEList l);`

Appends a list `l` to the end of a list given by its address `addr`. The address where the next element may be appended is returned. The list is modified as a side-effect of this function.

`int ElemInTYPEList (TYPE e, TYPEList l, TYPECmpFctType cmpfct);`

This function returns 1 iff the element `e` is in the list `l`. List elements are compared by the function `cmpfct`.

`TYPEList AddToSetTYPEList (TYPE e, TYPEList l, TYPECmpFctType cmpfct)`

A list is returned that has `e` as an element. `l` is check whether it already contains `e` using the compare function `cmpfct`.

`TYPEList MapTYPEList (TYPEList l, TYPEMapFct f);`

Returns a new list obtained by applying `f` to each element of `l`.

`int CompTYPEList (TYPEList l1, TYPEList l2, TYPECmpFctType f);`

Compares the lists `l1` and `l2` lexicographically by applying `f` to the corresponding elements.

`TYPE SumTYPEList (TYPEList l, TYPESumFct f, TYPE a);`

Applies the binary function `f` to the elements of the list: `f( f(... f(a, e1), e2, ...), en)` If `l` is empty `a` is returned.

It should be pointed out that the functions `AppElTYPEList`, `InsertAfterTYPEList`, `OrderedInsertTYPEList`, `RefEndConsTYPEList`, `RefEndAppTYPEList` modify existing lists and hence cause side-effects. If the non modified original list values are still to be used they have to be copied (`CopyTYPEList`) before they are modyfied. The other functions can be used in a strictly functional style.

8.3 Bit Sets of Arbitrary Length

This module implements operations on sets over elements which are nonnegative numbers. The range of each set value is dynamically adapted as reqired by the operations.

Storage for set values is allocated when needed. The module implementation uses efficient dynamic storage allocation of the `obstack`. The module does not implement automatic garbage collection. Storage used by one instance of this module can be deallocated completely, or for each single set value.

Some set operations are provided in two versions: The functional version allocates a new result value and leaves its operands unchanged. The imperative version modifies one of its operands to represent the result of the operation. Note: In the imparative case, too, the result of the function call rather than the operand has to be used for subsequent accesses to the modified set value, e. g. `s = AddElemToBitSet (x, s);` where s is a `BitSet` variable.

The module does not have generic parameters. It is used by writing

```
$/Adt/BitSet.fw
```

in a `.reqmod` (or a `.specs`) file.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `BitSet.h` is imported there.

The module exports the following type names and macros:

`BitSet` A pointer type representing sets.

`NullBitSet`
: The 0 pointer representing an empty set. Note: The empty set can be represented in different ways. Hence, the function `EmptyBitSet` must be used to check for a set being empty.

The following set processing functions are supplied by the module:

`void FreeBitSet (BitSet s)`
: Deallocates the set `s`.

`void FreeMemBitSet (void)`
: Deallocates all memory allocated for sets.

`BitSet NewBitSet (void)`
: Allocates an empty set.

`int EqualBitSet (BitSet s1, BitSet s2)`
: Yields 1 if `s1` and `s2` contain the same elements; otherwise 0.

`int EmptyBitSet (BitSet s)`
: Yields 1 if `s` is empty; otherwise 0.

`int EmptyIntersectBitSet (BitSet s1, BitSet s2)`
: Yields 1 if the intersection of `s1` and `s2` is empty; otherwise 0.

`int ElemInBitSet (int el, BitSet s)`
: Yields 1 if `el` is an element of `s`; otherwise 0.

`int CardOfBitSet (BitSet s)`
: Yields the number of elements in `s`.

`BitSet AddElemToBitSet (int el, BitSet s)`
: Imperative: Adds element `el` to set `s`.

`BitSet AddRangeToBitSet (int el1, int el2, BitSet s)`
: Imperative: All elements in the range from `el1` to `el2` are added to the set `s`.

`BitSet SubElemFromBitSet (int el, BitSet s)`
: Imperative: Subtracts element `el` from set `s`.

`BitSet UnionToBitSet (BitSet s1, BitSet s2)`
: Imperative: `s1` is set to the union of `s1` and `s2`

`BitSet IntersectToBitSet (BitSet s1, BitSet s2)`
: Imperative: `s1` is set to the intersection of `s1` and `s2`.

`BitSet SubtractFromBitSet (BitSet s1, BitSet s2)`
: Imperative: `s2` is subtracted from `s1`.

`BitSet ComplToBitSet (int upb, BitSet s)`
: Imperative: `s` is complemented with respect to the range 0 .. `upb`; no assumption can be made on elements larger than `upb` in `s`

`BitSet UniteBitSet (BitSet s1, BitSet s2)`
: Functional: Yields the union of `s1` and `s2`

`BitSet IntersectBitSet (BitSet s1, BitSet s2)`

Functional: Yields the intersection of `s1` and `s2`

`BitSet SubtractBitSet (BitSet s1, BitSet s2)`

Functional: Yields `s1` minus `s2`.

`BitSet ComplBitSet (int upb, BitSet s)`

Functional: Yields the complement of `s` with respect to the range 0 .. `upb`; no assumption can be made on elements larger than `upb` in the result.

`int NextElemInBitSet (int elem, BitSet s)`

Yields the smallest element of `s` that is larger than `elem`, if any; -1 otherwise.

`void ApplyToBitSet (BitSet s, void func(int))`

Applies the function `func` to each element of `s`

`void PrintBitSet (BitSet s)`

Prints `s` as a string of 0 and 1 to `stdout`.

`void PrintElemsBitSet (BitSet s)`

Prints `s` as a comma separated sequence of its elements to `stdout`.

8.4 Bit Sets of Integer Size

This C module implements sets of small nonnegative integral values by values of an `unsigned` type that is determined on instantiation of the module. The maximal element value depends on the number of bits used for the representation of the chosen `unsigned` type. The operations provided by this module may be used for computations on kind set values (or any other suitable application). In the following description of the set operations we assume that `el` stands for an expression that yields an integer being a set element, and `s, s1, s2` stand for expressions yielding sets.

The module is instantiated by

```
$/Adt/IntSet.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `IntSet` instance and `TYPE` specifies the type used for representing set values. If the `referto` parameter is specified to be `TYPE`, then `unsigned TYPE` has to be a valid integral C type. Examples for the `referto` parameter are `short`, `int`, or `'long int'`. If the `referto` parameter is omitted `int` is assumed. The `instance` parameter may be omitted if there is only one instance of this module.

Note: The interface exported by this module is contained in the file `NAMEIntSet.h`. If it is to be used in other specifications that file has to be included in a `.head` specification.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `IntSet.h` is imported there.

The module exports the following type names and macros (all names are prefixed by the value `NAME` of the `instance` parameter):

`NAMEIntSet`
: the type representing sets.

`NAMENULLIS`
: the unique value for an empty set.

`NAMENullIS()`
: a macro with no arguments for the unique value for an empty set, to be used where a functional notation is required as in `WITH` clauses of `CONSTITUENTS`.

The following operations for set processing are supplied by the module. It is indicated whether an operation is implemented by a function or by a macro. The macros do not evaluate any argument repeatedly.

`NAMEIntSet NAMESingleIS (int el)`
: Yields a singleton set containing `el`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEAddElIS (int el, NAMEIntSet s)`
: Yields the union of `NAMESingleIS(el)` and `s`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEConsIS (int el, NAMEIntSet s)`
: Yields the union of `NAMESingleIS(el)` and `s`; validity of `el` is not checked. (macro)

`int NAMEInIS (int el, NAMEIntSet s)`
: Yields 1 iff `el` is in `s`; a message is issued if `el` is not a valid element. (function)

`NAMEIntSet NAMEUniteIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields the union of `s1` and `s2`. (macro)

`NAMEIntSet NAMESubIS (NAMEIntSet s1, NAMEIntSet s2)`
: Yields `s1` minus `s2`. (macro)

`NAMEIntSet NAMEInterIS (NAMEIntSet s1, NAMEIntSet s2)`

Yields the intersection of `s1` and `s2`. (macro)

`int NAMEDisjIS (NAMEIntSet s1, NAMEIntSet s2)`

Yields 1 iff `s1` and `s2` are disjoint. (macro)

`int NAMEInclIS (NAMEIntSet s1, NAMEIntSet s2)`

Yields 1 iff `s1` is a subset of `s2`. (function)

`int NAMEEqualIS (NAMEIntSet s1, NAMEIntSet s2)`

Yields 1 iff `s1` is equal to `s2`. (macro)

`int NAMEEmptyIS (NAMEIntSet s)`

Yields 1 iff `s` is empty. (macro)

`int NAMECardIS (NAMEIntSet s)`

Yields the number of elements in `s`. (function)

8.5 Stacks of Any Type

This module implements a stack named `NAMEStack`, whose elements are of type `TYPE`. Values of this type can be pushed onto the stack and popped off of it in the usual way, and in addition each element of the stack can be indexed directly and its value obtained. All of the operations exported by this module are implemented as macros, using the facilities of the obstack module (see Section 'Memory Object Management' in Library Reference Manual).

The module is instantiated by

```
$/Adt/Stack.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `Stack` instance and `TYPE` is the element type.

Note: The interface exported by this module is contained in the file `NAMEStack.h`. If it is to be used in `.lido` specifications that file has to be included in a `.head` specification.

The following functions are supplied by the module:

`int NAMEStackEmpty`

Yields the value 1 if the stack has no elements, 0 otherwise.

This operation must appear in an expression context.

`void NAMEStackPush(TYPE v)`

Push an element onto the stack. The parameter `v` must be an expression that yields a value of type `TYPE`. That value becomes the new top element of the stack. The previous top element becomes the new second element, and so on.

This operation must appear in a statement context.

`TYPE NAMEStackPop`

Remove the top element of the stack. The previous second element becomes the new top element, and so on.

This operation must appear in an expression context, and yields the value (of type `TYPE`).

`TYPE NAMEStackTop`

Obtain the contents (of type `TYPE`) of the top element of the stack without changing the state of the stack.

This operation must appear in an expression context.

`TYPE NAMEStackElement (int i)`

Obtain the contents (of type `TYPE`) of a specific element of the stack without changing the stack. The argument gives the distance of the desired element from the top of the stack (0 for the newest element, 1 for the next newest, and so on). There is no check on the validity of the value of `i`.

This operation must appear in an expression context.

`TYPE NAMEStackArray (int i)`

Obtain the contents (of type `TYPE`) of a specific element of the stack without changing the state of the stack. For the purpose of this operation, the stack is considered to be an array. Element 0 is the oldest value on the stack, element 1 is the next oldest, and so on. There is no check on the validity of the value of `i`.

This operation must appear in an expression context.

`void ForEachNAMEStackElementDown (int i)`

Cycle through the elements of the stack, from the most recent to the oldest. The parameter `i` must be declared as an lvalue of type `TYPE*` and will point, in turn, to each element of the stack.

This operation must appear in a context where `for (i=...; i>=...;i--)` is allowed.

`void ForEachNAMEStackElementUp (int i)`

Cycle through the elements of the stack, from the oldest to the most recent. The parameter `i` must be declared as an lvalue of type `|`KEY`|*` and will point, in turn, to each element of the stack.

This operation must appear in a context where `for (i=...; i<...;i++)` is allowed.

8.6 Mapping Integral Values To Other Types

This module implements mappings from non-negative integers to an arbitrary type that is specified by a generic instantiation parameter. Any assignable type can be chosen.

The map is implemented by a dynamically allocated array using the `obstack` module via the `DynSpace` interface. The mapping storage can be deallocated.

The size of the mapping has to to be stated on initialization.

The module is instantiated by

```
$/Adt/Map.gnrc +instance=NAME +referto=TYPE :inst
```

where `NAME` identifies the `Map` instance and `TYPE` is the target type of the mapping.

Note: The target type of the mapping `TYPE` must be either a type that is predefined in C, or its definition must be made available by some `.head` specification.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `NAMEMap.h` is imported there.

The following functions are supplied by the module:

`void NAMEInitMap (int max_n)`
: Initializes the mapping for value between 0 and `max_n`.

`void NAMEFinlMap (void)`
: Deallocates the mapping.

`void NAMEInitMapValues (TYPE v)`
: Initializes all mappings in the range 0 to `max_n` to the value `v`.

`void NAMESetMap (int n, TYPE v)`
: Maps `n` to the value `v`. It may override a previously set value.

`TYPE NAMEGetMap (int n)`
: Yields the `TYPE` value set for `n` by the most recent call of `NAMESetMap`, if any; otherwise the result is undefined.

8.7 Dynamic Storage Allocation

This module provides functions for dynamic storage allocation. Its operations are significantly faster than the C function `malloc`, since storage is allocated within larger chunks which are less frequently requested from the system. The module is implemented on top of a more general storage allocation module `obstack`, which may be used directly if more elaborate allocation mechanisms are needed.

The use of the module requires an initializing call of the function `InitDynSpace`.

`void* InitDynSpace ()`
: initializes a storage area for subsequent allocations. The result should be assigned to a variable used in subequent allocation calls.

The module does not have generic parameters. It is used by writing

```
$/Adt/DynSpace.fw
```

in a `.reqmod` (or a `.specs`) file.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in C modules if the interface file `DynSpace.h` is imported there.

The module exports the following functions:

`void* InitDynSpace (void)`
: Initializes a dynamic storage area and returns a pointer to it. it has to be used in any subsequent call of `DynAlloc`.

`void* DynAlloc (void *space, int size)`
: Allocates storage of the given `size` in the area pointed to by `space`.

`void DynClear (void *space)`
: Deallocates the complete storage area pointed to by `space`.

The functions can be used in a C module in the following way:

```
#include "DynSpace.h"
void *MySpace;
/* ... */
```

```
MySpace = InitDynSpace ();
/* ... */
p = (TypeX*) DynAlloc (MySpace, sizeof (TypeX));
/* ... */
DynClear (MySpace);
/* ... */
```

Note: Neither the cast nor the call of sizof can be part of a LIDO computation, since type identifiers are not allowed in computations. Hence the calls of this module must reside in a C module (that implements the type of the allocated objects).

9 Solutions of Common Problems

This library contains the following modules:

`Message` Error Reports (module deleted)

`Strings` String Concatenation

`Counter` Counting Symbol Occurrences

`MakeName` Generating Optional Identifiers

9.1 Error Reports

This module is NOT supported anymore.

Uses of the two macros `Message` and `MessageId` that were provided by this module have to be rewritten, since LIGA does NOT allow to hide accesses of `COORDREF`.

A macro call `Message(s, t)` can be rewritten `message(s, t, 0, COORDREF)`.

A macro call `MessageId(s, t, sym)` can be rewritten

```
message(s, CatStrInd(t, sym), 0, COORDREF)
```

where `CatStrInd` is a macro that stores the concatenation of two strings. It is provided by the module (see Section 9.2 [Strings], page 111):

```
$/Tech/Strings.specs
```

9.2 String Concatenation

This module provides functions for concatenation of strings, and the name `CharPtr` for the type `char *`.

The module is instantiated by

```
$/Tech/Strings.specs
```

The module defines the type name `CharPtr` for `char *`, especially to be used in LIDO specifications.

The module exports the functions

`CharPtr CatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and the pointer to it is returned.

`int IndCatStrStr (CharPtr s1, CharPtr s2)`
: The strings `s1, s2` are concatenated. The resulting string is stored in the string memory of the `csm` module, and its `StringTable` index is returned.

A macro `CatStrInd(s,i)` is exported it obtains the second of the concatenated strings by the index `i` into the `StringTable`. This macro is used to simplify composition of `message` texts.

9.3 Counting Symbol Occurrences

This module provides `.lido` computations that count all ocurrences of certain symbols within a certain subtree.

The module is instantiated by

```
$/Tech/Counter.gnrc +instance=NAME :inst
```

The optional generic instance parameter `NAME` identifies the particular instance.

The module provides the following computational roles:

`NAMECount` is associated to the grammar symbols that shall be counted. `NAMECount.NAMECount` yields the occurrence number in the tree (starting from 1).

`NAMERootCount` specifies the subtree containing the `NAMECount` occurrences. The number of occurrences found in the subtree can be obtained by the attribute `NAMERootCount.NAMECountResult`.

`NAMERootCount` can be associated to recursive grammar symbols. Any symbol that has `NAMECount` associated must belong to a subtree such that `NAMERootCount` is associated to its root.

9.4 Generating Optional Identifiers

This C module implements functions that turn their arguments into strings which then play the role of identifiers as if they occurred in the input, i.e. they are entered in the identifier table, and their symbol code is returned.

The module is instantiated by

```
$/Tech/MakeName.gnrc +instance=IDENT :inst
```

The generic instance parameter `IDENT` has to be set to the terminal symbol used for identifiers.

The module can also be used to generate and store identifiers in processors that do not have a scanner and a parser. In this case the `+instance` parameter has to be omitted.

All entities exported by this module can be used in specifications of type `.lido`, `.init`, `.finl`, and `.con`. They can also be used in `.pdl` specifications or in C modules if the interface file `MakeName.h` is imported there.

The module exports the following functions:

`int MakeName (char *c)`
: `c` is a character string that may coincide with an already existing name. The result the is the encoding of the name `c`.

`int GenerateName (char *c)`
: `c` is a character string. The name is generated by appending a number to the prefix `c` such that the name is different from all others encountered so far (on input and generated).

`int IdnNumb (int id, int num)`
: `id` is the encoding of an existing identifier; `num` is a nonnegative number. The new name is formed by appending the number to the identifier string. It may coincide with an already existing name. This function may be used to derive arbitrary names from existing ones.

`int PreIdnPost (char *pre, int id, char *post)`

`id` is the encoding of an existing identifier, `pre` and `post` are arbitrary character strings. The generated name is formed by catenation of `pre`, the identifier string and `post`. It may coincide with an already existing name. This function may be used to derive different names (e. g. for different target objects) from an existing name.

10 Migration of Old Library Module Usage

This section helps to migrate applications of previous module library version to the current one.

Users who want to continue to use library modules of a previous Eli version, please contact your Eli system manager.

`ModLib3_8`

Migration of Eli Version 3.8 modules

`ModLib3_6`

Migration of Eli Version 3.6 modules

10.1 Migration of Eli Version 3.8 modules

This section gives a quick reference to modifications that have been made in library modules of Eli version 3.8. Users who are still using modules of Eli version 3.6 first see Section 10.2 [ModLib3˙6], page 118.

In all library modules that provide LIDO specifications the symbol roles are now specified to be `CLASS SYMBOL`s. This classification avoids accidental name coincidences between grammar symbols and roles provided by modules.

As a consequence such a symbol may not occur as a name of a tree grammar symbol. It rather has to be inherited to a tree grammar symbol:

```
RULE: Program ::= Source END;
SYMBOL Program INHERITS RootScope END;
```

Constructs like

```
RULE: RootScope ::= Source END;
```

will cause error messages. Hence, the grammar symbols have to be renamed consistently in such cases.

We assume that some names like `IdDef` or `IdUse` have been used in this now illegal way. In order to simplify migration those module roles are now renamed `IdDefScope` and `IdUseEnv`. Hence, specifications which use `IdDef` or `IdUse` as grammar symbols may continue to do so. Only

```
SYMBOL IdDef INHERITS IdDefScope END;
SYMBOL IdUse INHERITS IdUseEnv END;
```

is to be added.

The following table contains a list symbol roles that are provided by modules of Eli version 3.8 and have been changed in the current version. All these adaptions refer to modules of the `Name` library.

The following library modules have been modified:

`BitSet` Use `$/Adt/BitSet.fw` instead of instantiation. (see Section 8.3 [BitSet], page 102)

`DynSpace` Use `$/Adt/DynSpace.fw` instead of instantiation. (see Section 8.7 [DynSpace], page 109)

`LeafPtg` Use `$/Output/LeafPtg.fw` instead of instantiation. (see Section 7.1 [LeafPtg], page 87)

`Indent` Use `$/Output/Indent.fw` instead of instantiation. (see Section 7.2 [Indent], page 88)

`List` The module uses `obstack` directly instead of using module `DynSpace`.

`OutStr` Use `$/Output/OutStr.fw` instead of instantiation. (see Section 7.3 [OutStr], page 89)

`Name Analysis`

There are minor modifications in all name analysis modules. See table of entity modifications below, and (see Section 3.2 [Basic Scope Rules], page 13) (see Section 3.11 [Scope Properties], page 29) (see Section 3.15 [Inheritance of Scopes], page 35)

`PreDefId` Instead of instantiating the module `PreDefId` once for each predefined identifier, all predefined identifiers are be described in one file. Its name is supplied as `referto` parameter. (see Section 3.6 [Predefined Identifiers], page 22)

`PropLib` Use `$/Prop/PropLib.fw` instead of instantiation. (see Section 4.10 [PropLib], page 56)

`Type CType BuType`

These modules are still usable. But they are not maintained anymore. It is recommended to use the module `Typing` (See Section 5.1 [BasicType], page 60) instead. The functionality of `CType` and `BuType`, where the C-like name analysis scheme is also imposed upon type analysis is not supported by `Typing`. Modules having corresponding facilities are not yet available.

The following entities of library modules have been modified:

`AnyScope.GotScopes`
: Replaced: use `RootScope.GotScopeProp` (see Section 3.11 [Scope Properties], page 29)

`AnyScope.GotScopesDefs`
: Replaced: use `RootScope.GotScopeProp` (see Section 3.11 [Scope Properties], page 29)

`IdDef`
: Renamed to `IdDefScope`. (see Section 3.2 [Basic Scope Rules], page 13)

`IdDefScopeProp`
: Deleted: The scope property is now associated to a key in the context of the role `RangeScopeProp`. (see Section 3.11 [Scope Properties], page 29)

`IdGetScopeProp`
: Deleted: Use the `GetScope` function explicitly if necessary. (see Section 3.11 [Scope Properties], page 29)

`IdUse`
: Renamed to `IdUseEnv`. (see Section 3.2 [Basic Scope Rules], page 13)

`IdUseScope`
: modified: The default scope to be used can not be changed (see Section 3.2 [Basic Scope Rules], page 13); use `IdUseScopeProp` if necessary (see Section 3.11 [Scope Properties], page 29).

`InheritScopeProp`
: Renamed and modified: use `InheritScope`. (see Section 3.15 [Inheritance of Scopes], page 35)

`RangeScopeProp`
: Modified: A computation of `THIS.ScopeKey` for the key to which the scope is to be associated has to be provided, instead of `THIS.Scope`. (see Section 3.11 [Scope Properties], page 29)

10.2 Migration of Eli Version 3.6 modules

The following table is a quick reference for migration of module usage from module library upto Eli version 3.6 to modules of this library. For each module name of the old library a reference to the documentation of the corresponding module of this library is given. In many cases it is sufficient to modify the instantiation command. Others require modifications of identifiers in `.lido` specifications, or in the way they are used.

Note: The modules of the library of Eli version 3.6 which are mentioned below are no longer contained in the Eli distribution. Users who want to continue to use them, please contact your Eli system manager.

`AdaptOil` use Section 5.8 [Operator], page 75

`Bool` removed

`Chain` use Section 3.4 [CScope], page 18

`ChainPtg` removed, see PTG documentation

`CmdLineIncl`
now in Chapter 6 [Input], page 81 library, See Section 6.3 [CmdLineIncl], page 84

`CoordMap` now in Chapter 6 [Input], page 81 library, See Section 6.3 [CmdLineIncl], page 84

`CurrTok` now in Chapter 6 [Input], page 81 library, See Section 6.2 [CurrTok], page 83

`DefPt` use Section 4.3 [SetFirst], page 49 in Chapter 4 [Prop], page 43 library

`DynSpace` now in Chapter 8 [Adt], page 97 library, See Section 8.7 [DynSpace], page 109

`Field` use Section 3.12 [AlgScopeProp], page 31 or Section 3.13 [CScopeProp], page 32

`Fwd` use Section 4.7 [Kind], page 52 or Section 4.8 [KindSet], page 53 in Chapter 4 [Prop], page 43 library

`GenName` use Section 9.4 [MakeName], page 113

`GChain` use Section 3.5 [BuScope], page 20

`InclLido` use Section 6.1 [Include], page 81 in Chapter 6 [Input], page 81 library

`Include` use Section 6.1 [Include], page 81 in Chapter 6 [Input], page 81 library

`Indent` now in Chapter 7 [Output], page 87 library, See Section 7.2 [Indent], page 88

`IntSet`

`IntToKey` use Section 8.6 [Map], page 108 in Chapter 8 [Adt], page 97 library

`IntToPtr` use Section 8.6 [Map], page 108 in Chapter 8 [Adt], page 97 library now in Chapter 8 [Adt], page 97 library, See Section 8.4 [IntSet], page 104

`KindSet` unchanged, See Section 4.8 [KindSet], page 53

`LeafPtg` now in Chapter 7 [Output], page 87 library, See Section 7.1 [LeafPtg], page 87

`List` use Section 8.1 [LidoList], page 97 in Chapter 8 [Adt], page 97 library

`Message` use Section 9.2 [Strings], page 111 to compose message texts and call the `message` function directly, as described in Section 9.1 [Message], page 111.

`Nest` use Section 3.3 [AlgScope], page 16

`NoKeyMsg` removed, See Section 3.2 [Basic Scope Rules], page 13

`OccCnt` unchanged, See Section 4.2 [OccCnt], page 47

`Once` use Section 4.5 [FirstOcc], page 51

`OutStr` now in Chapter 7 [Output], page 87 library, See Section 7.3 [OutStr], page 89

`PreDef` use `PreDefine` and `PreDefId`, See Section 3.6 [Predefined Identifiers], page 22

`PtrList` use Section 8.2 [List], page 99 in Chapter 8 [Adt], page 97 library

`Stack` now in Chapter 8 [Adt], page 97 library, See Section 8.5 [Stack], page 106

`Unique` now in Chapter 4 [Prop], page 43 library, See Section 4.4 [Unique], page 50

Index

A

B

R

S

T

U

V