Type analysis tasks

Uwe Kastens

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 1.12 $

Table of Contents

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 yield when it is evaluated. 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.

This document has the following structure:

The section (See Chapter 1 [BasicType], page 3) 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 Chapter 2 [TypeProp], page 9) gives examples how to use the `BasicType` module for languages having user defined types (e.g. `array` types). The section (See Chapter 3 [Operator], page 18) 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 Chapter 4 [Defer], page 23) 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 `DefTableKeys`. Hence, properties can be associated to a type to further characterize the type, as described in (See Chapter 2 [TypeProp], page 9). `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 named `Type` are associated to symbols describing the type of a program construct.

A complete executable specification of our running example for the type analysis task can be obtained by calling Eli and requesting

```
$elipkg/Type/Examples > .
```

This request creates a subdirectory `Examples` in your current working directory containing the example specification `Type.fw`, test input files, and an Odinfile for automatic regression testing.

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 the following section, which intorduces a running example. A complete executable specification for an example is available in `$/Type/Examples/Type.fw`.

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 supplies the key 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 Chapter 4 [Defer], page 23) 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. Let `k` be a type identifier; then a call `TransDefer (k)` accesses the type key at the end of the `Defer` chain via arbitrary many steps.

This module provides the following computational roles:

`RootType`: is inherited by the grammar root by default.

`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 `TypedDefinition`, `TypedDefId`, `TypedUseId`, and `TypeDefUseId` are used. Those are sufficient for languages that have only predefined types. User defined types require additionally 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. It guarantees that the property values can be accessed where the typed object is used.

An attribute `SYNT.TypeIsSet` represents the state where the `TypeOf` property of this `TypedDefId` has been set. The default computations of `TypedDefId` and `TypedUseId` are set up such that the

`TypeOf` properties of all `TypedDefId` are set before any `TypeOf` property of a `TypedUseId` is accessed. In cases where a different dependency pattern is needed, it can be obtained by overriding the computations of `SYNT.TypeIsSet` in the two contexts. An example for a different dependency pattern is a left-to-right chain which may allow to set the type in a defining occurrence using applied occurrences that have been set before.

`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. See the explanation of the attribute `SYNT.TypeIsSet` for the role `TypedDefId`.

`ChkTypedUseId` checks that the key represents a typed object, and issues an error message otherwise.

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 an 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 is inherited by the grammar root by default. 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.

1.1 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.fw.

For the moment we assume the there are only three predefined types int, float, and bool. They are represented by the keys intType, realType, and boolType as introduced in (see Section 'Predefined Identifiers' in Name analysis according to scope rules).

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

Our language construct `ObjDecl` defines typed objects. It specifies that the `Type` attribute is 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.Type;
END;

ATTR Type: DefTableKey;
```

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

The types of literal expressions are 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 adjacent 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 particular language. (An example implementation is shown in `$/Type/Examples/Type.fw`).

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 Chapter 2 [TypeProp], page 9) it is demonstrated how to introduce some common types.

The `TypeDefIdent` has the role of a 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;
```

2 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 Chapter 1 [BasicType], page 3) and some modules of the property library (see Section 'Property Library' in Association of properties to definition).

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.

2.1 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 Chapter 1 [BasicType], page 3). 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;
```

2.2 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.fw`.

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 'Some Useful PDL Specifications' in Association of properties to definitions, which is automatically instantiated when using the basic type module.

2.3 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 'Lists in LIDO Specifications' in Abstract data types to be used in specifications).

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:      [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 'Lists in LIDO Specifications' in Abstract data types to be used in specifications,):

```
$/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.fw`.

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
```

2.4 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 'Scopes Being Properties of Objects' in Name analysis according to scope rules, 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 'Scope Properties Algol-like' in Name analysis according to scope rules, 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 'Scope Properties Algol-like' in Name analysis according to scope rules), 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 RootScope.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.

3 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 `DefTableKeys` 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 `DefTableKeys`.

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
```

If this instantiation is contained in a `.specs` file and if the description file, say `Oper.d` is contained in the same directory, it may read

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

If the `.specs` and `Oper.d` are contained in a `.fw` specification, the same line can be used.

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 `DefTableKeys`. 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 by 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 by 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 UnOprs 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, (intType,intType):intType,    TgtStr={"+"})
TgtOpr (AddKey, bOrKey,  (boolType,boolType):boolType, TgtStr={"||"})
TgtOpr (MulKey, iMulKey, (intType,intType):intType,    TgtStr={"*"})
TgtOpr (MulKey, bAndKey, (boolType,boolType):boolType, TgtStr={"&&"})
TgtOpr (NegKey, iNegKey, (intType):intType,            TgtStr={"-"})
TgtOpr (NegKey, bNotKey, (boolType):boolType,          TgtStr={"!"})
```

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

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

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 ")"
```

4 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 keys 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 'Some Useful PDL Specifications' in Association of properties to definitions,) 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 by 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.

`RootDefer` is inherited by the grammar root by default.

Index