Type Analysis Reference Manual

Uwe Kastens
W. M. Waite

University of Paderborn
D-33098 Paderborn
Germany

Table of Contents

1 Types, Operators, and Indications

A *type* characterizes a subset of values in the universe manipulated by the program, an *operator* defines an operation applied to operands of specified types to produce a result of a specific type, and an *indication* defines a set of operators. These three concepts form the basis for any type model. The designer must specify language-defined types, operators, and indications; it may also be possible for the user to provide additional specifications as part of a program (see Chapter 4 [User-Defined Types], page 23).

Although language-defined types may be specified individually, they are usually introduced by language-defined operator specifications. These, along with language-defined indication specifications, are described in a language called *OIL* (see Section "OIL's Specification Language" in *Oil Reference Manual*). OIL text is written in a file whose name has the form '`name`'`.oil`.

1.1 Language-defined types

Each type is represented by a unique definition table key whose `IsType` property has the value 1. Further properties of that key may be used to provide information about that particular type. `NoKey` represents an unknown or non-existent type.

Language-defined types like "integer" are represented by known keys (see Section "Initializations" in *Definition Table*). The known key name can be used directly in an attribute computation. For example, suppose that the designer chose `intType` as the name of the known key for the Java integer type. The following rule would then interpret the keyword `int` as denoting that type:

```
RULE: Type ::= 'int' COMPUTE
  Type.Type=intType;
END;
```

Language-defined types are sometimes denoted by pre-defined identifiers (as in Pascal) instead of keywords. This approach increases the complexity of the specification by introducing type identifiers (see Section 4.2 [Type identifiers], page 24). It also allows a user to re-define the names of language-defined types as names of variables or parameters, making programs very hard to understand. We recommend that designers use keywords to denote language-defined types.

1.2 Language-defined operators

An operator has a fixed signature, and is represented by a unique definition table key. Properties of that key may be used to provide information about that particular operator. `NoKey` represents an unknown or non-existent operator.

The OIL statement `OPER 'opr' 'sig';` defines the key and signature of an operator:

'`opr`' The name of the known definition table key representing the operator. Multiple operator definitions with the same value of '`opr`' are not allowed.

'`sig`' The signature of the operator represented by '`opr`'. It consists of a parenthesized (possibly empty), comma-separated list of operand types followed by a colon and a return type. All of the types in the signature are automatically defined as known keys representing types; no further specification is required.

Only one occurrence of the keyword `OPER` is required for a sequence of contiguous operator definitions:

```
OPER
  iAddOp (intType,intType):intType;
  fAddOp (floatType,floatType):floatType;
  iGtrOp (intType,intType):boolType;
```

The known keys named `intType`, `floatType`, and `boolType` are defined by this specification, each with its `IsType` property set to 1. Known keys named `iAddOp`, `fAddOp`, and `iGtrOp` are also defined; no further specification of these names is necessary.

Often there are a number of language-defined operators sharing the same signature. A shorthand notation for describing such a situation allows the designer to provide a comma-separated list of operator names and write the shared signature only once:

```
OPER
  iAddOp,iSubOp,iMulOp,iDivOp (intType,intType):intType;
  fAddOp,fSubOp,fMulOp,fDivOp (floatType,floatType):floatType;
```

1.3 Language-defined indications

Each operator must belong to a set of operators associated with some indication, also represented by a unique definition table key. Properties of that key may be used to provide additional information about that indication. `NoKey` represents an unknown or non-existent indication.

The OIL statement `INDICATION 'ind': 'list';` defines a subset of the operators associated with an indication:

'`ind`' The known definition table key representing the indication. Multiple indication definitions with the same value of '`ind`' are allowed. In that case, the operator set associated with the indication is the union of the sets specified by the individual definitions.

'`list`' A comma-separated list of operators in the indication's set.

Only one occurrence of the keyword `INDICATION` is required for a sequence of contiguous indication definitions:

```
INDICATION
  PlusInd: iAddOp, fAddOp;
  MinusInd: iSubOp, fSubOp;
```

1.4 Language-defined coercibility

Language properties like the "usual arithmetic conversions" of C and the "widening conversions" of Java allow the compiler to accept an operand of one type as though it were a value of another type. We use the relation `acceptableAs` on types to model these properties. `acceptableAs` is a partial order:

Reflexive ('`T`' `acceptableAs` '`T`') for any type '`T`'

Transitive ('`T`' `acceptableAs` '`T1`') and ('`T1`' `acceptableAs` '`T2`') for some types '`T`', '`T1`', and '`T2`' implies ('`T`' `acceptableAs` '`T2`')

Antisymmetric
: (‘T’ acceptableAs ‘T1’) and (‘T1’ acceptableAs ‘T’) implies ‘T’ is identical to ‘T1’

To see why these properties are important, consider the following expression in C or Java (s is of type short and f is of type float):

```
s + f
```

Both C and Java allow implicit conversion of short to int and int to float in the context of an arithmetic operand. Thus a designer would specify (short acceptableAs int) and (int acceptableAs float) for C or Java. Transitivity guarantees that (short acceptableAs float), and reflexivity guarantees that (float acceptableAs float), so the operator fAddOp can be selected from the set associated with the indication PlusInd of the last section.

Suppose that (float acceptableAs int). In that case, the meaning of the expression is ambiguous. There is no way to decide whether to select the operator iAddOp or the operator fAddOp from PlusInd's set. But because acceptableAs is antisymmetric, (float acceptableAs int) would imply that int and float were identical types. Thus the designer cannot specify (float acceptableAs int) for C or Java.

The acceptableAs relation is specified by defining *coercion* operators. The OIL statement COERCION ‘opr’ ‘sig’; defines the key and signature of a coercion:

‘opr’
: The name of the known definition table key representing the coercion operator. If ‘opr’ is omitted, the OIL compiler will generate a unique name internally. Multiple coercion definitions with the same value of ‘opr’ are not allowed. A coercion definition cannot have the same value of ‘opr’ as an operator definition.

‘sig’
: The signature of the coercion operator represented by ‘opr’. It consists of a parenthesized operand type followed by a colon and a return type. Both types in the signature are automatically defined as known keys representing types; no further specification is required.

Only one occurrence of the keyword COERCION is required for a sequence of contiguous coercion operator definitions:

```
COERCION
  sToi (shortType):intType;
       (intType):floatType;
```

This specification illustrates both named and anonymous coercions. Generally speaking, coercions need be named only if they are to be discussed in associated documentation or extracted to support further processing (such as the evaluation of constant expressions).

1.5 Reducing specification size

A full specification of language-defined operators often leads to a combinatorial explosion. In many applications the effects of this explosion on the written specification can be mitigated by avoiding unnecessary operator names. For example, the task of type analysis is to verify type correctness; the identity of the operator that models the type behavior at a specific node is normally irrelevant.

Language definitions avoid combinatorial explosions by giving names to sets of types and then defining properties of operations in terms of these sets rather than the individual elements. For example, the C definition describes operations on "arithmetic types" rather than describing those operations on integers and then again on floating-point values. OIL provides a notation for naming and manipulating sets of types that allows the designer to encode such language definitions directly.

The OIL statement `SET 'name' = 'expr' ;` defines a set of types:

'`name`' An identifier naming a set. Multiple sets with the same name are not allowed.

'`expr`' An expression defining the types that are members of the set. There are five possible expression formats:

`[ 'elements' ]`
: Each member of the comma-separated list '`elements`' is a known key representing a type. That type is an element of the value of this expression. There are no other elements.

'`name`' The previously-defined set '`name`' is the value of this expression.

`'s1' + 's2'`
: The value of this expression is the union of set '`s1`' and set '`s2`'.

`'s1' * 's2'`
: The value of this expression is the intersection of set '`s1`' and set '`s2`'.

`'s1' - 's2'`
: The value of this expression is the set of elements of '`s1`' that are not elements of '`s2`'.

Here are some definitions that mirror the C language standard:

```
SET Signed_IntegerType =
  [signed_charType, shortType, intType, longType];

SET Unsigned_IntegerType =
  [unsigned_charType, unsigned_shortType,
   unsigned_intType,  unsigned_longType];

SET FloatingType =
  [floatType, doubleType, long_doubleType];

SET IntegralType =
  [charType] + Signed_IntegerType + Unsigned_IntegerType;

SET ArithmeticType = IntegralType + FloatingType;

SET ScalarType = ArithmeticType + [VoidPointerType];
```

A specific context in a program will often require a value that can be of any type in a particular set. For example, the condition value in a C `if` statement or conditional expression can be of any scalar type. We model this situation by defining a "type" (`scalarType`, say) and making each scalar type acceptable as that type. The context can then require a value of `scalarType`, and any scalar type will be acceptable.

When a type set name is used in an `OPER` or `COERCION` signature, the result is a number of distinct operators. Each operator's signature is constructed by consistently substituting one element of the named type set for each instance of the type name. Thus every scalar type can be made acceptable as `scalarType` as follows:

```
COERCION (ScalarType):scalarType;
```

Similarly, signatures containing type set names can be used to reduce the number of specifications needed for operators. For example, consider the following specification:

```
OPER ArithOp (ArithmeticType, ArithmeticType): ArithmeticType;
```

It defines a set of 12 operators, each named by the known key `ArithOp`. Each operator has a distinct signature, one of which is `(charType,charType):charType`. That signature results from the consistent substitution of the `charType` element of `ArithmeticType` for the name of that set in the `OPER` statement's signature.

This set of 12 operators can be associated with an indication:

```
INDICATION ArithInd: ArithOp;
```

Because the same element of a type set is substituted for each instance of the name of that set in a signature, the only way to get all combinations of elements is to create another name for that set and use both names in the signature. For example, a value of any scalar type in C can be cast to any other scalar type:

```
SET CastResult = ScalarType;
OPER ScalarCast (ScalarType):CastResult;
```

One of the operators named by the known key `ScalarCast` has the signature `(charType):floatType`. That signature results from substituting the `charType` element of `ScalarType` for the name of that set and the `floatType` element of `CastResult` for the name of that set.

2 Typed Entities

A *typed entity* is a named program component, one of whose properties is a type. Variables, formal parameters, and fields are the most common typed entities; functions are also typed entities in some languages (see Section 5.3 [Functions as typed entities], page 32). When an identifier is used to represent a typed entity, the type specified by a defining occurrence of that identifier must be made available at each applied occurrence. This is accomplished through the use of a `DefTableKey`-valued property of the definition table key characterizing the typed entity.

The `Typing` module exports computational roles to implement the definition and use of typed entities:

`TypedDefinition`
: The computational role inherited by a grammar symbol that represents a definition of one or more typed entities having the same type.

`TypedDefId`
: The computational role inherited by a grammar symbol that represents a defining occurrence of an identifier for a typed entity.

`TypedUseId`
: The computational role inherited by a grammar symbol that represents an applied occurrence of an identifier for a typed entity.

The `Typing` module is instantiated by

```
$/Type/Typing.gnrc +referto='prefix' :inst
```

`'prefix'Key` (or simply `Key` if the `referto` parameter is missing) must be the name of an attribute of every grammar symbol inheriting the `TypedDefId` or `TypedUseId` role. The value of that attribute must be the definition table key bound to the symbol during name analysis (see Section "Name analysis according to scope rules" in *Specification Module Library*).

2.1 Establishing the type of an entity

A typical local variable declaration from Java or C specifies a type and a list of variable names:

```
int a, b, c;
```

The entire declaration plays the role of a `TypedDefinition`; each of `a`, `b`, and `c` plays the role of a `TypedDefId`.

The value of the `TypedDefinition.Type` attribute must be set by a user computation to the definition table key of the type. No other user computations are needed because default computations provided by the `Typing` module in descendant `TypedDefId` constructs will access the `TypedDefinition.Type` attribute, setting the appropriate property of the definition table key characterizing the typed entity.

A Java or C compiler might use the following specification to describe a variable declaration:

```
SYMBOL VrblDecl INHERITS TypedDefinition END;
SYMBOL VarIdDef INHERITS TypedDefId      END;

RULE: VrblDecl ::= Type VarIdDefs ';' COMPUTE
  VrblDecl.Type=Type.Type;
END;

RULE: VarIdDefs LISTOF VarIdDef END;
```

2.2 Accessing the type of an entity

TypedUseId is an applied occurrence of an identifier representing a typed entity. A Typing module computation sets the value of the TypedUseId.Type attribute to the definition table key representing the entity's type.

If ExpIdUse represented an applied occurrence of a variable or parameter identifier in the abstract syntax tree, the Typing module will provide a value for ExpIdUse.Type if the following line appears in the specification:

```
SYMBOL ExpIdUse INHERITS TypedUseId END;
```

2.3 Passing ancillary information

The Typing module guarantees that every TypedUseId.Type attribute depends on all of the type analysis computations (see Section 7.4 [Dependences for typed entities], page 42). In other words, any computation accessing TypedUseId.Type is guaranteed to take place after all type analysis computations have been completed. This dependence can be used to guarantee the availability of information characterizing the typed entity that is ancillary to type analysis.

The operation that sets the Type property of the definition table key characterizing the typed entity depends on the void attribute TypedDefId.GotProp. TypedDefId.GotProp is set by an accumulating computation that can be augmented by an upper-context accumulating computation of the symbol inheriting TypedDefId or by an accumulating computation in a rule having the symbol inheriting TypedDefId on the right-hand side. The Typing module will then guarantee that all such a computations have been carried out before any access to TypedUseId.Type is allowed. Any such computations must, however, be independent of all results of type analysis.

Pascal's distinction between variable and value parameters is a typical example of information ancillary to type analysis that must be conveyed from defining to applied occurrences of typed entities:

```
ATTR IsVarParam: int;

SYMBOL FormalParamSect INHERITS TypedDefinition COMPUTE
  SYNT.IsVarParam=0;
END;

RULE: FormalParamSect ::= 'var' Formals ':' Type COMPUTE
  FormalParamSect.IsVarParam=1;
END;

SYMBOL FormalIdDef INHERITS TypedIdDef COMPUTE
  INH.GotProp+=
    ResetIsVarParam(THIS.Key,INCLUDING FormalParamSect.IsVarParam);
END;

SYMBOL ExpIdUse INHERITS TypedUseId COMPUTE
  SYNT.IsVarParam=GetIsVarParam(THIS.Key,0) <- THIS.Type;
END;
```

This computation assumes that an integer-valued property `IsVarParam` has been defined. It is set by a computation in the upper context of `FormalIdDef` that augments the default computation of the void attribute `TypedIdDef.GotProp`, and queried by a symbol computation in the lower context of `ExpIdUse`. The latter computation depends on `ExpIdUse.Type`, so the `Typing` module guarantees that the property value has been set for every formal parameter before it is queried.

`FormalParamSect.IsVarParam` is an integer valued attribute, distinct from the `IsVarParam` property, set by a symbol computation in the lower context of `FormalParamSect`. That symbol computation is overridden in the rule representing a declaration of a variable parameter.

Note that a particular instance of symbol `ExpIdUse` in the tree does not necessarily represent an applied occurrence of a formal parameter. (It might represent an applied occurrence of a variable identifier, for example.) Thus the `IsVarParam` property might not be set; the query will return the default value 0 in that case. The overall effect of these computations is therefore to set the value of `ExpIdUse.IsVarParam` to 1 if and only if that instance of `ExpIdUse` represents an applied occurrence of a variable parameter.

3 Expressions

An *expression node* represents a program construct that yields a value, and an *expression tree* is a subtree of the abstract syntax tree made up entirely of expression nodes. Type analysis within an expression tree is uniform; additional specifications are needed only at the roots and leaves. (Note that these need not be roots and leaves in the sense of the abstract syntax tree.) A designer often chooses to represent a programming language expression by more than one expression tree, in order to capture special relationships within that expression. For example, each argument of a function call might be a separate expression tree because more type conversions are allowed in that context than in the context of an operator.

The `Expression` module provides computational roles and rule computations to implement the type analysis of expression trees:

`ExpressionSymbol`
: The computational role inherited by a grammar symbol that represents an expression node.

`OperatorSymbol`
: The computational role inherited by a grammar symbol that represents an operator node.

`OpndExprListRoot`
`BalanceListRoot`
: Computational roles inherited by grammar symbols that represent operand lists.

`OpndExprListElem`
`BalanceListElem`
: Computational roles inherited by grammar symbols that represent operand list elements. Every `OpndExprListElem` must be a descendant of `OpndExprListRoot` in the tree; every `BalanceListElem` must be a descendant of `BalanceListRoot`.

`PrimaryContext`
`TransferContext`
`BalanceContext`
`MonadicContext`
`DyadicContext`
`ListContext`
`ConversionContext`
`CastContext`
`RootContext`
: Rule computations implementing common expression contexts.

`Indication`
`OperName`
: Rule computations for expression contexts where operations are performed, but which have no grammar symbol representing the possible operations.

The expression module is usually instantiated by:

```
$/Type/Expression.gnrc :inst
```

For a discussion of alternatives, see Section 3.2 [Selecting an operator at an expression node], page 13.

3.1 Type analysis of expression trees

The symbol on the left-hand side of a rule defining an expression node characterizes the expression's result. It inherits the `ExpressionSymbol` role. Two attributes of `ExpressionSymbol` describe its type:

`Required` An inherited attribute whose `DefTableKey` value represents the type required by the surrounding context. (A value of `NoKey` indicates that no specific type is required.) `Required` may be set by a user computation at the root of an expression subtree; computations are supplied by the `Expression` module for all other expression nodes.

`Type` An attribute whose `DefTableKey` value is set by computations supplied by the `Expression` module to represent the type of the result delivered by the expression subtree rooted in this node. (A value of `NoKey` indicates that the type delivered by the node is unknown.) `Type` may depend on `Required` as well as on the possible types of the node's children; it must never be set by user computation.

An expression node is type-correct if the type specified by its `Type` attribute is acceptable as the type specified by its `Required` attribute. Any type is acceptable as an undefined required type, and an undefined type is acceptable as any required type.

In order to support incremental development, `ExpressionSymbol` defines default computations setting the values of both `Required` and `Type` to `NoKey`; those computations are overridden by the rule computations described in this chapter. The default computations allow one to declare that a symbol inherits `ExpressionSymbol` without producing specification errors for every context containing that symbol. This advantage is offset by the fact that if one forgets to provide rule computations for some contexts, the generated compiler will silently ignore certain errors in the input program.

Rules defining expression nodes in an abstract syntax tree for a typical programming language describe constants, variables, and computations:

```
SYMBOL Expr INHERITS ExpressionSymbol   END;

RULE: Expr   ::= Number                 END;
RULE: Expr   ::= ExpIdUse               END;
RULE: Expr   ::= Expr Operator Expr     END;
RULE: Expr   ::= Expr '[' Subscript ']' END;
```

The first two rules describe leaves of expression subtrees. Any node described by the first rule is a leaf of the abstract syntax tree as well as a leaf of some expression subtree. Nodes described by the second rule are not leaves of the abstract syntax tree because each has an `ExpIdUse` node as a child.

A leaf of an expression subtree delivers a value whose type must be determined from the context of that leaf according to the definition of the language. For example, the `Expr` node in the first rule might deliver the language-defined integer type; in the second rule, the delivered type is the value of `ExpIdUse.Type`.

The type analyzer models most interior expression nodes by operators applied to operands:

1. An indication is derived from the context.
2. One operator is selected from the set associated with that indication.
3. The `Type` attribute of the node is set to the result type of the selected operator.
4. The `Required` attributes of one or more children are set to the operand types of the selected operator.

For example, in the following rule, the indication is provided by the `Operator` child:

```
RULE: Expr   ::= Expr Operator Expr     END;
```

Usually, a set of several operators (such as {`iAddOp`, `fAddOp`}) is associated with that indication. An operator is then selected from that set as discussed in the next section.

In the fourth rule, we might assume that each array type definition adds a dyadic access operator to an indication fixed by the rule (see Section 4.4 [Operator definitions], page 26):

```
RULE: Expr   ::= Expr '[' Subscript ']' END;
```

The left operand of that operator is the array type, the right operand is the index type, and the result is the element type.

The operator/operand model provides support for expression node semantics that are ubiquitous in programming languages. Several other models, useful in special circumstances, are supported and will be introduced in later sections of this chapter. It is clear, however, that there will be situations in which the semantics of an expression context do not fit any of the supported models. Our advice is to consider such a context as a place where several disjoint expression subtrees meet: The expression symbol on the left-hand side of the rule defining the context is a leaf of an expression tree above the context, and each expression symbol on the right-hand side is the root of an expression tree below the context.

3.2 Selecting an operator at an expression node

If an indication is associated with a singleton operator set, that operator is selected regardless of operand or result types.

There are two standard algorithms for selecting an operator if the indication's set has more than one element. The simplest ignores the type required by the context. For each operator in the set, it checks whether each operand is acceptable as the type required by that operator. If more than one operator in the set satisfies this condition, the algorithm chooses the one requiring the fewest coercions. If no operator can be identified, then the unknown operator is chosen.

To select this algorithm, instantiate the `Expression` module without the `+referto` parameter:

```
$/Type/Expression.gnrc :inst
```

Ada is a language in which the selection of an operator depends on the type required by the context as well as the types delivered by the operands. In that case, a two-pass algorithm is required.

Starting with the leaves of an expression tree and working towards the root of that tree, the algorithm determines a *possible type set* for each expression node. Every type in the possible type set at a node is either a leaf type or is a type associated with an operator whose result is that type, and whose operands are elements of the possible type sets of the node's children. The algorithm associates a *cost* with each type in a possible type set.

OIL allows one to specify an arbitrary integer cost for each operator and coercion independently. If this specification is omitted, a cost of 1 is assigned to the operator or coercion. For the remainder of this section, we assume that all cost specifications have been omitted and therefore all costs are 1.

Consider a very simple expression node, the integer constant 3. One element of the possible type set for this node is the language-defined integer type, which has cost 0 because no operations are needed to create a value of that type. If a coercion has been defined with the language-defined integer type as its operand and the language-defined floating-point type as its result, then another element of the possible type set of this node is the language-defined floating-point type. It has cost 1, the total number of operators required to produce it. Similarly, if there is a coercion from floating-point to double, double will be an element of the possible type set of the node and it will have cost 2.

When the algorithm computes the possible type set of an interior expression node, it considers the operators in that node's indication set. For each operator, it checks whether the type required by that operator for a given argument is in the possible type set of the corresponding operand. Each operator meeting that condition is a possible operator at the node. The cost of using an operator is one more than the sum of the costs associated with its argument types in the possible type sets of its operands. Finally, the possible type set of the node is the set of all types `T` such that the result type of a possible operator is acceptable as `T`.

Often a particular element of the possible type set of an interior node can be obtained in more than one way. For example, consider a node representing the sum of two integers. Assuming that the integer type is acceptable as the floating-point type, the possible type set of each child contains both integer and floating-point types. Thus both integer addition and floating-point addition are possible selections at this node. There are then two ways to obtain a floating-point result:

1. Use an integer addition and convert the result to floating-point.
2. Convert each operand to floating-point and use a floating-point addition.

The cost of using an integer addition in this context is 1 because the cost of the integer elements of the possible type sets are both 0. Converting the result to floating-point costs one coercion, for a total cost of 2. The cost of using a floating-point addition, on the other hand, is 3 because the cost of the floating-point elements of the possible type sets are both 1.

The cost of obtaining a value of type `T` using a particular operator is the sum of the cost of using that operator and the number of coercion operators required to convert a value of the result type to a value of type `T`. When more than one possible operator selection leads to a value of a given type, the algorithm only retains the one with the lowest cost.

If the `Required` attribute of the expression tree root is the unknown type, then the algorithm chooses the lowest-cost element of the root's possible type set as the result type of the expression. Otherwise, the `Required` attribute of the root is taken as the result type regardless of whether it appears in the root's possible type set.

The second pass starts with the root of the tree and works towards the leaves. At each node, the value of the `Required` attribute specifies an element of the possible type set and hence an operator. Given an operator, the values of the node's `Type` attribute and the `Required` attributes of any operands are fixed.

If the possible type set of an expression node does not contain an element equal to the value of that node's `Required` attribute, then the unknown operator is selected; the node's `Type` attribute and the `Required` attributes of any operands are set to the unknown type.

To select the two-pass algorithm, instantiate the `Expression` module with the `+referto=Result` parameter:

```
$/Type/Expression.gnrc +referto=Result :inst
```

3.3 Expression contexts without operators

Let '`e1`' be a grammar symbol playing the `ExpressionSymbol` role, and '`type`' be an expression yielding the definition table key of a type. A *primary context* is one in which the parent '`e1`' delivers a value of a known type. `PrimaryContext('e1','type')` provides the rule computations to set '`e1`'`.Type` to the type '`type`'. (Recall that the value of the `Type` attribute of an expression node must never be set directly by a user computation.)

The constant and variable expressions in C are examples of primary contexts:

```
SYMBOL Expr INHERITS ExpressionSymbol END;

RULE: Expr ::= Number COMPUTE
  PrimaryContext(Expr,intType);
END;

SYMBOL ExpIdUse INHERITS TypedUseId END;

RULE: Expr ::= ExpIdUse COMPUTE
  PrimaryContext(Expr,ExpIdUse.Type);
END;
```

The type of an integer constant is the language-defined integer type (see Section 1.1 [Language-defined types], page 1), and the type of a variable is the type with which it was declared (see Section 2.2 [Accessing the type of an entity], page 8).

Let '`e1`' and '`e2`' be grammar symbols playing the `ExpressionSymbol` role. A *transfer context* is one in which the parent '`e1`' and one of its children '`e2`' are identical with respect to type. `TransferContext('e1','e2')` provides the rule computations to set '`e1`'`.Type` and '`e2`'`.Required`.

The comma expression in C is an example of a transfer context:

```
RULE: Expr ::= Expr ',' Expr COMPUTE
  TransferContext(Expr[1],Expr[3]);
END;
```

Notice that the left operand of the comma, `Expr[2]`, is the root of an expression subtree distinct from the one containing the `TransferContext`. The value of this expression will be discarded, so its type is arbitrary. Thus there is no need to override the default computation `Expr[2].Required=NoKey`.

Let ‘e1’, ‘e2’, and ‘e3’ be grammar symbols playing the ExpressionSymbol role. A *balance context* is one in which the parent ‘e1’ must deliver either the result delivered by child ‘e2’ or the result delivered by child ‘e3’. This means that values delivered by both children must be acceptable as a common type, and the parent must deliver a value of that common type. BalanceContext(‘e1’,‘e2’,‘e3’) provides the rule computations to set ‘e1’.Type, ‘e2’.Required, and ‘e3’.Required such that the following relations hold:

- ‘e2’.Type acceptableAs ‘e1’.Type
- ‘e3’.Type acceptableAs ‘e1’.Type
- There is no type ‘t’ other than ‘e1’.Type such that
  - ‘e2’.Type acceptableAs ‘t’
  - ‘e3’.Type acceptableAs ‘t’
  - ‘t’ acceptableAs ‘e1’.Type
- ‘e2’.Required equals ‘e1’.Type
- ‘e3’.Required equals ‘e1’.Type

The conditional expression of C is an example of a balance context:

```
RULE: Expr ::= Expr '?' Expr ':' Expr COMPUTE
  BalanceContext(Expr[1],Expr[3],Expr[4]);
  Expr[2].Required=scalarType;
END;
```

The condition, Expr[2], is the root of an expression subtree distinct from that containing the BalanceContext. The definition of C requires that Expr[2] return a value of scalar type, independent of the types of the other expression nodes. (Pointers and numbers are values of scalar type in C.) Thus the default computation Expr[2].Required=NoKey must be overridden in this context.

Some languages generalize the conditional expression to a case expression. For example, consider an ALGOL 68 computation of the days in a month:

```
begin int days, month, year;
days := case month in
  31,
  (year mod 4 and year mod 100 <> 0 or year mod 400 = 0 | 28 | 29),
  31,30,31,30,31,31,30,31,30,31 esac
end
```

The number of cases is not fixed, and the balancing process therefore involves an arbitrary list. This is the purpose of the BalanceListRoot and BalanceListElem roles. Both inherit from ExpressionSymbol, and neither requires computations beyond the ones used in any expression context. The balancing computation described above is carried out pairwise on the list elements:

```
SYMBOL CaseExps INHERITS BalanceListRoot END;
SYMBOL CaseExp  INHERITS BalanceListElem END;

RULE: Expr ::= 'case' Expr 'in' CaseExps 'esac' COMPUTE
  TransferContext(Expr[1],CaseExps);
END;

RULE: CaseExps LISTOF CaseExp END;

RULE: CaseExp ::= Expr COMPUTE
  TransferContext(CaseExp,Expr);
END;
```

Notice that these rule computations simply interface with the `BalanceListRoot` and `BalanceListElem` roles; all significant computations are done by module code generated from those roles.

3.4 Operators with explicit operands

Tree symbols in the abstract syntax that correspond to operator symbols in a source program usually inherit the `OperatorSymbol` role. Two attributes of `OperatorSymbol` describe the operator selection in the current context:

`Indic`
: A synthesized attribute whose `DefTableKey` value represents the indication derived from the context. (A value of `NoKey` indicates that no such indication can be derived.) `Indic` must be set by a user computation.

`Oper`
: An attribute whose `DefTableKey` value is set by `Expression` module computations to represent the operator selected from the set identified by the associated indication. (A value of `NoKey` indicates that no operator could be selected.) `Oper` may depend on `Required` as well as on the possible types of the node's children and the operator indication; it must never be set by user computation.

In order to support incremental development, `OperatorSymbol` defines a default computation setting the values of both `Indic` and `Oper` to `NoKey`. The default computation of `Indic` is overridden by a user computation, and that of `Oper` by the rule computations described in this section. The default computations allow one to declare that a symbol inherits `OperatorSymbol` without producing specification errors for every context containing that symbol. This advantage is offset by the fact that if one forgets to provide appropriate overriding computations, the generated compiler will silently ignore certain errors in the input program.

Let '`e1`', '`e2`', and '`e3`' all play the `ExpressionSymbol` role, and '`rator`' play the `OperatorSymbol` role. A *monadic (dyadic) context* is one in which the parent '`e1`' delivers the result of applying '`rator`' to the operand(s). The following provide rule computations to set '`rator`'`.Oper`, '`e1`'`.Type`, and '`e2`'`.Required` (plus '`e3`'`.Required` if present):

- `MonadicContext('e1','rator','e2')`
- `DyadicContext('e1','rator','e2','e3')`

Contexts with arbitrary numbers of operands are discussed in the next section.

```
SYMBOL Operator INHERITS OperatorSymbol END;

RULE: Expr ::= Expr Operator Expr COMPUTE
  DyadicContext(Expr[1],Operator,Expr[2],Expr[3]);
END;

RULE: Operator ::= '+' COMPUTE
  Operator.Indic=PlusInd;
END;
```

The array access rule also fits the `DyadicContext` pattern, but has no symbol playing the `OperatorSymbol` role. In such cases, the '`rator`' argument is omitted and the indication supplied by an additional context-dependent rule computation.

Let '`ind`' be a definition table key representing an indication. `Indication('ind')` provides the rule computations to set the node's indication to '`ind`'.

If the indication `indexInd`'s operator set includes one access operator for every array type (see Section 4.4 [Operator definitions], page 26), then the following computation implements the type relationship in an array access:

```
SYMBOL Subscript INHERITS ExpressionSymbol END;

RULE: Expr ::= Expr '[' Subscript ']' COMPUTE
  DyadicContext(Expr[1],,Expr[2],Subscript);
  Indication(indexInd);
END;
```

Note that `Expr[2]` can be *any* expression yielding an array value; it need not be a simple array name.

In some cases it is useful to know the name of the operator selected from the indication set. The `OperatorSymbol.Oper` attribute normally supplies this information, but when there is no symbol playing that role the value can be accessed via a context-dependent rule computation:

`OperName` Yields the operator selected from the context's indication set. If no operator can be selected, the result is the unknown operator.

3.5 Operators with operand lists

Function calls and multidimensional array references are common examples of expression contexts whose operators have operand lists rather than explicit operands. One symbol on the right-hand side of the rule defining such a context characterizes the entire list of operands. It inherits the `OpndExprListRoot` role.

The symbol defining an operand in the list inherits the `OpndExprListElem` role. `OpndExprListElem` inherits the `ExpressionSymbol` role, and overrides `ExpressionSymbol`'s default computation of the `Required` attribute in all upper contexts.

Let '`e`' be a grammar symbol playing the `ExpressionSymbol` role, '`rator`' be a grammar symbol playing the `OperatorSymbol` role, and '`rands`' be a grammar symbol playing the

`OpndExprListRoot` role. A *list* context is one in which the parent '`e`' delivers the result of applying '`rator`' to the operand list '`rands`'. `ListContext('e','rator','rands')` provides the rule computations to set '`e`'`.Type`, '`rator`'`.Oper`, and `OpndExprListElem.Required` for each `OpndExprListElem` descendant of '`rands`'.

If the language has multi-dimensional array references, they can be implemented using a strategy that differs from that of the previous section:

```
SYMBOL Subscripts INHERITS OpndExprListRoot END;
SYMBOL Subscript  INHERITS OpndExprListElem END;

RULE: Expr ::= Expr '[' Subscripts ']' COMPUTE
  ListContext(Expr[1],,Subscripts);
  Indication(GetAccessor(Expr[2].Type,NoKey));
END;

RULE: Subscripts LISTOF Subscript END;

RULE: Subscript ::= Expr COMPUTE
  TransferContext(Subscript,Expr);
END;
```

This computation assumes that the indication is the value of the `Accessor` property of the array type (see Chapter 4 [User-Defined Types], page 23).

Some languages have *variadic* operators – operators taking a variable number of operands. The most common of these are `max` and `min`, which can take two or more numeric operands. All of the operands must ultimately be of the same type, so the situation is similar to that of a balanced context.

For type checking purposes, the variadic operator can be considered to have a single operand, whose type is determined by balancing the elements of the list (see Section 3.3 [Expression contexts without operators], page 15). Of course this form of operand list must be distinguished syntactically from a normal list operand:

```
SYMBOL VarRands INHERITS BalanceListRoot END;
SYMBOL VarRand  INHERITS BalanceListElem END;

RULE: Expr ::= VarOper '(' VarRands ')' COMPUTE
  MonadicContext(Expr,VarOper,VarRands)
END;

RULE: VarRands LISTOF VarRand END;

RULE: VarRand ::= Expr COMPUTE
  TransferContext(Actual,Expr);
END;
```

3.6 Type conversion

The `acceptableAs` relation models implicit type conversion in the context of operators applied to operands. In other contexts, additional type conversions may be possible. For example, both Java and C allow a floating-point value to be assigned to an integer variable. That conversion cannot be modeled by the `acceptableAs` relation (see Section 1.4 [Language-defined coercibility], page 2).

Additional type conversions such as those taking place on assignment can be modeled by specific conversion operators. An indication is associated with each context in which additional type conversions are possible, and the indication's set contains exactly the conversions allowable in that context.

Let '`e1`' and '`e2`' play the `ExpressionSymbol` role, and '`rator`' play the `OperatorSymbol` role. A *conversion context* is one in which the rules of the language allow the type conversions in '`rator`'`.Indic`'s set to be applied to the value yielded by '`e2`' (in addition to any coercions) in order to obtain the type that must be yielded by '`e1`'. `ConversionContext('e1','rator','e2')` provides rule computations to set '`rator`'`.Oper`, '`e1`'`.Type`, and '`e2`'`.Required`. If no additional conversion operator is required, or if none can be selected from '`rator`'`.Indic`'s set, then '`rator`'`.Oper` is set to the unknown operator and both '`e1`'`.Type` and '`e2`'`.Required` are set to '`e1`'`.Required`.

In C, an actual argument to a function call may be implicitly converted to the type of the corresponding formal parameter prior to the function call. The same set of conversions can be used in assignment contexts, so assume that the indication is `assignCvt`:

```
SYMBOL Actual INHERITS OpndExprListElem END;

RULE: Actual ::= Expr COMPUTE
  ConversionContext(Actual,,Expr);
  Indication(assignCvt);
END;
```

Let '`e1`' and '`e2`' play the `ExpressionSymbol` role, '`rator`' play the `OperatorSymbol` role, and '`type`' yield a `DefTableKey` value representing a type. A *cast context* is a conversion context in which the desired type is inherent in the context itself, rather than being determined by '`e1`'. `CastContext('e1','rator','e2','type')` provides rule computations to set '`rator`'`.Oper`, '`e1`'`.Type`, and '`e2`'`.Required`. If no additional conversion operator is required, or if none can be selected from '`rator`'`.Indic`'s set, then '`rator`'`.Oper` is set to the unknown operator and both '`e1`'`.Type` and '`e2`'`.Required` are set to '`type`'.

The C cast expression is an example of a cast context. Here we assume that `castInd` is an indication whose set consists of all of the possible C conversions:

```
RULE: Expr ::= '(' Type ')' Expr COMPUTE
  CastContext(Expr[1],,Expr[2],Type.Type);
  Indication(castInd);
END;
```

Let '`e2`' play the `ExpressionSymbol` role, '`rator`' play the `OperatorSymbol` role, and '`type`' yield a `DefTableKey` value representing a type. A *root context* is a conversion context in which the desired type is inherent in the context itself, which is not an expression context.

RootContext(‘type’,‘rator’,‘e2’) provides rule computations to set ‘rator’.Oper and ‘e2’.Required. If no additional conversion operator is required, or if none can be selected from ‘rator’.Indic’s set, then ‘rator’.Oper is set to the unknown operator and ‘e2’.Required is set to ‘type’.

The C return statement is an example of a root context. It is not itself an expression, but it has an expression operand. That operand must yield the return type of the function, which is inherent in the context of the return statement, and can be obtained from the Function node. Here we assume that assignInd is an indication whose set consists of all of the possible C assignment conversions:

```
RULE: Statement ::= 'return' Expr COMPUTE
  RootContext(INCLUDING (Function.ResultType),,Expr);
  Indication(assignInd);
END;
```

4 User-Defined Types

A language that permits user-defined types must provide constructs for the user to denote such types. These constructs are called *type denotations*. If a programmer writes two type denotations that look the same, it is natural to ask whether they represent the same type. There are two general answers to this question:

Name equivalence
: Each type denotation that the programmer writes represents a distinct type.

Structural equivalence
: Two type denotations represent the same type if they are constructed in the same way and if corresponding components are the same (see Chapter 5 [Structural Type Equivalence], page 31).

All of the techniques discussed in this document apply independently of the selection of name equivalence or structural equivalence among user-defined types.

A *type identifier* is a name used in a source language program to refer to a type. It is important to distinguish between the concept of a type and the concept of a type identifier, using different keys to implement them, because a particular type might have zero or more type identifiers referring to it. For example, consider the following snippet of C code:

```
typedef float time;
typedef float distance;
typedef struct { time t; distance d; } leg;
leg trip[100];
```

This snippet creates two user-defined types, a structure type and an array (or pointer) type. Moreover, it defines three type identifiers, `time`, `distance`, and `leg`. The first two refer to the language-defined float type, and the third refers to the structure type; the array type is *anonymous* — no type identifier refers to it. Seven definition table keys are therefore associated with the types and type identifiers of this snippet; three more are associated with the typed entities `t`, `d`, and `trip` (see Chapter 2 [Typed Entities], page 7).

The `Typing` module exports computational roles to implement the definition and use of user-defined types:

`TypeDenotation`
: The computational role inherited by a grammar symbol that represents a subtree denoting a type.

`TypeDefDefId`
: The computational role inherited by a grammar symbol that represents a defining occurrence of a type identifier.

`TypeDefUseId`
: The computational role inherited by a grammar symbol that represents an applied occurrence of a type identifier.

4.1 Type denotations

Type denotations are language constructs that describe user-defined types. The symbol on the left-hand side of a rule defining a type denotation characterizes the type denoted. It inherits the `TypeDenotation` role, which provides three attributes:

`Type` A `DefTableKey`-valued attribute representing the type denoted by this subtree. This attribute is set by a module computation that should never be overridden by the user. It should be used in any computation that does not require properties of the type.

`TypeKey` A `DefTableKey`-valued attribute representing the type denoted by this subtree. This attribute is set by a module computation that should never be overridden by the user. It should be used in any computation that accesses properties of the type.

`GotType` A void attribute representing the fact that information characterizing a user-defined type has been stored as properties of the key `TypeDenotation.Type`.

The information stored as properties of the definition table key `TypeDenotation.Type` cannot be dependent on the results of type analysis (see Section 7.4 [Dependences for typed entities], page 42).

For example, some languages (e.g. Modula-3, Ada) allow a user to define a subrange type that is characterized by its bounds. The bound information may be needed in various contexts where the type is used, and therefore it is reasonable to store that information as properties of the subrange type's key. Suppose, therefore, that `Lower` and `Upper` are defined as integer-valued properties. Bound information is independent of any aspect of type analysis:

```
SYMBOL SubrangeSpec INHERITS TypeDenotation END;
RULE:  SubrangeSpec ::= '[' Number 'TO' Number ']' COMPUTE
  SubrangeSpec.GotType+=
    ORDER(
      ResetLower(SubrangeSpec.Type,atoi(StringTable(Number[1]))),
      ResetUpper(SubrangeSpec.Type,atoi(StringTable(Number[2]))));
END;
```

Here `Number` is a non-literal terminal symbol whose value is the digit string appearing in the source text; `atoi` is the string-to-integer conversion routine from the C library.

4.2 Type identifiers

The computational role `TypeDefDefId` is inherited by a defining occurrence of a type identifier. It provides two attributes:

`Type` A `DefTableKey` value representing the type named by the type identifier. This attribute must be set by a user computation. It should be used in any computation that does not require properties of the type.

`TypeKey` A `DefTableKey`-valued attribute representing the type denoted by this subtree. This attribute is set by a module computation that should never be overridden by the user. It should be used in any computation that accesses properties of the type.

The computational role `TypeDefUseId` is inherited by an applied occurrence of a type identifier. It provides two attributes:

`Type` A `DefTableKey` value representing the type named by the type identifier. This attribute is set by a module computation that should never be overridden by the user. It should be used in any computation that does not require properties of the type.

`TypeKey` A `DefTableKey`-valued attribute representing the type denoted by this subtree. This attribute is set by a module computation that should never be overridden by the user. It should be used in any computation that accesses properties of the type.

4.3 Referring to a type

A type might be referenced in program text in any of three different ways, each illustrated by a Java or C variable definition:

1. By writing a keyword, as in `int v;`
2. By writing a type identifier, as in `t v;`
3. By writing a type denotation, as in `struct {int i; float f;} v;`

Each of these representations of a type uses its own mechanism for encoding the type. In order to standardize the encoding, a type reference is normally represented in the tree by a distinct symbol having a `DefTableKey`-valued `Type` attribute (see Section 2.1 [Establishing the type of an entity], page 7). For example, `Type` plays that role in this representation for a variable declaration:

```
RULE: VrblDecl ::= Type VarIdDefs ';' COMPUTE
  VrblDecl.Type=Type.Type;
END;
```

Here the value of `Type.Type` represents some type. That attribute must be defined by providing a rule establishing the type represented by a type identifier, a rule establishing each language-defined type represented by a keyword, and a rule establishing each user-defined type represented by a type denotation:

```
SYMBOL TypIdUse INHERITS TypeDefUseId END;

RULE: Type ::= TypIdUse COMPUTE
  Type.Type=TypIdUse.Type;
END;

RULE: Type ::= 'int' COMPUTE
  Type.Type=intType;
END;

RULE: Type ::= SubrangeSpec COMPUTE
  Type.Type=SubrangeSpec.Type;
END;
```

The `Type` attributes discussed in this chapter generally do not give direct access to properties of the type they represent, because many of their values are intermediate in the type analysis computations (see Section 7.1 [Dependences among types and type identifiers], page 40). If it is necessary to access properties of a type at a symbol inheriting `TypeDenotation`, `TypeDefDefId` or `TypeDefUseId`, use the `TypeKey` attribute. Values of the `Type` attribute of a symbol inheriting `ExpressionSymbol` or `TypedUseId` can be used directly to access type properties.

4.4 Operator, function, and method definitions

A user-defined type is often associated with one or more operators. For example, an array type requires an access operator (see Section 3.4 [Operators with explicit operands], page 17). The `Expression` module provides computational roles and rule computations to define these operators:

`OperatorDefs`
: The computational role inherited by a grammar symbol that represents a context where operators are defined.

`OpndTypeListRoot`
`OpndTypeListElem`
: Computational roles inherited by grammar symbols that represent operand definition lists.

`MonadicOperator`
`DyadicOperator`
`ListOperator`
`Coercible`
: Rule computations implementing definition contexts.

All operators associated with user-defined types must be added to the database of valid operators before type analysis of expressions can begin. This dependence is made explicit by having the left-hand side symbol of any rule in which operators are defined inherit the `OperatorDefs` role. One attribute is used to express the dependence:

`GotOper`
: A void attribute indicating that *all* of the operator definitions in this rule have been carried out. It is set by a module computation that should be overridden by the user.

The `OpndTypeListRoot` role is inherited by a grammar symbol representing a list of operand types. It has one attribute:

`OpndTypeList`
: A synthesized attribute whose `DefTableKeyList` value is a list of the operand types in reverse order. It is set by a module computation that should not be overridden by the user.

The `OpndTypeListElem` role is inherited by a grammar symbol representing a single operand type in a list. It must be a descendant of a node playing the `OpndTypeListRoot` role, and has one attribute:

`Type`
: A synthesized attribute whose `DefTableKey` value is set by user computation to represent the operand type.

Operators are actually defined by rule computations. Let '`ind`', '`opr`', '`rand`', '`rand1`', '`rand2`', and '`rslt`' be definition table keys and '`rands`' be a list of definition table keys.

`MonadicOperator('ind','opr','rand','rslt')`
: Adds operator `'opr'('rand'):'rslt'` to the set named by indication '`ind`'.

`DyadicOperator('ind','opr','rand1','rand2','rslt')`
: Adds operator `'opr'('rand1','rand2'):'rslt'` to the set named by indication '`ind`'.

`ListOperator('ind','opr','rands','rslt')`
: Adds operator `'opr'(t1,...,tn):'rslt'` to the set named by indication '`ind`'. Here `t1,...,tn` are the values obtained from '`rands`'.

`Coercible('opr','rand','rslt')`
: Adds coercion `'opr'('rand'):'rslt'` to the coercions in the database.

The actual value of '`opr`' is often irrelevant in these computations, because the designer does not ask which operator was selected from the given indication. The `Expression` module provides the known key `NoOprName` for use in these situations.

Consider a type denotation for one-dimensional arrays. Assume that a subscript must be of the language-defined integer type, and that each new array type overloads the standard array indexing indication `indexInd` with the indexing operator for that array (see Section 3.4 [Operators with explicit operands], page 17). The operator name is uninteresting:

```
SYMBOL ArraySpec INHERITS TypeDenotation, OperatorDefs END;

RULE: ArraySpec ::= Type '[' ']' COMPUTE
  ArraySpec.GotOper+=
    DyadicOperator(
      indexInd,
      NoOprName,
      ArraySpec.Type,
      intType,Type.Type);
END;
```

Another approach defines the `Accessor` property of the array type to be an indication with a singleton operator set (see Section 3.4 [Operators with explicit operands], page 17):

```
ATTR Indic: DefTableKey;

SYMBOL IndexTypes INHERITS OpndTypeListRoot END;

RULE: ArraySpec ::= Type '[' IndexTypes ']' COMPUTE
  .Indic=NewKey();
  ArraySpec.GotType+=ResetAccessor(ArraySpec.Type,.Indic);
  ArraySpec.GotOper+=
    ListOperator(
      .Indic,
      NoOprName,
      IndexTypes.OpndTypeList,
      Type[1].Type);
END;
```

Functions and methods are simply operators with operand lists. These operators overload the indication that is the function or method name. In many cases, of course, a singleton operator set will be associated with a function or method name. The operator name may or may not be interesting, depending on how the designer chooses to interpret the results of type analysis.

Java method definitions overload the method identifier:

```
SYMBOL MethodHeader INHERITS OperatorDefs     END;
SYMBOL Formals      INHERITS OpndTypeListRoot END;

RULE: MethodHeader ::= Type MethIdDef '(' Formals ')' COMPUTE
  MethodHeader.GotOper+=
    ListOperator(
      MethIdDef.Key,
      NoOprName,
      Formals.OpndTypeList,
      Type.Type);
END;
```

The corresponding method call uses the method identifier as the operator symbol in a list context. Its indication is its `Key` attribute, as in the declaration:

```
SYMBOL MethIdUse INHERITS OperatorSymbol COMPUTE
  SYNT.Indic=THIS.Key;
END;
SYMBOL Arguments INHERITS OpndExprListRoot END;

RULE: Expr ::= Expr '.' MethIdUse '(' Arguments ')' COMPUTE
  ListContext(Expr[1],MethIdUse,Arguments);
END;
```

Every value in a C enumeration is coercible to an integer:

```
SYMBOL enum_specifier INHERITS TypeDenotation, OperatorSymbol END;

RULE: enum_specifier ::= 'enum' '{' enumerator_list '}'
  enum_specifier.GotOper+=
    Coercible(NoOprName,enum_specifier.Type,intType);
END;
```

4.5 Reducing specification size

A user type definition often requires definition of a number of operators, based on the relationship between the new type and its components. Although all of those operations can be defined using the techniques of the previous section, it may be simpler to define a "template" for the particular type constructor and then instantiate that template at each corresponding type denotation.

The necessary information can be captured in an OIL class (see Section "Class definition" in *Oil Reference Manual*). For example, a set type in Pascal implies operators for union, intersection, membership, and comparison:

```
CLASS setType(baseType) BEGIN
  OPER
    setop(setType,setType): setType;
    setmember(baseType,setType): boolType;
    setrel(setType,setType): boolType;
  COERCION
    (emptyType): setType;
END;

INDICATION
  plus: setop;
  minus: setop;
  star: setop;
  in: setmember;
  equal: setrel;
  lsgt: setrel;
  lessequal: setrel;
  greaterequal: setrel;
```

Within the class definition, the class name (`setType` in this example) represents the type being defined. The parameters of the class (e.g. `baseType`) represent the related types. Thus a set requires a set member operation that takes a value of the base type and a value of the set type, returning a Boolean. Notice that the designer chose to use the same operator for union, intersection, and difference because all of these operators have the same signature and distinguishing them is irrelevant for type analysis.

Let '`cl`' be an OIL class name, and '`typ`', '`arg1`', '`arg2`', '`arg3`' be definition table keys representing types. Each of the following rule computations instantiates an OIL class with a specific number of parameters:

`InstClass0(c,typ)`
`InstClass1(c,typ,arg1)`
`InstClass2(c,typ,arg1,arg2)`
`InstClass3(c,typ,arg1,arg2,arg3)`

Create the operators defined by OIL class '`cl`' for type '`typ`'. Types '`arg1`', '`arg2`', and '`arg3`' are the parameters of the instantiation:

A class instantiation creates operators, so it should have the `GotOper` attribute as a postcondition:

```
SYMBOL TypeDenoter INHERITS TypeDenotation, OperatorDefs END;

RULE: TypeDenoter ::= 'set' 'of' type COMPUTE
  TypeDenoter.GotOper+=InstClass1(setType,TypeDenoter.Type,type.Type);
END;
```

5 Structural Type Equivalence

The specific rules governing structural equivalence of types vary greatly from one language to another. Nevertheless, their effect on the type analysis task can be described in a manner that is independent of those rules. That effect is embodied in the `StructEquiv` module, instantiated by

```
$/Type/StructEquiv.fw
```

5.1 Partitioning the set of types

This module defines two types as structurally equivalent if they satisfy two conditions:

1. They *might* be equivalent according to the language definition.
2. Corresponding components have equivalent types.

For example, consider the structure types in the following variable declarations:

```
struct a { int f; struct a *g; } x;
struct b { int h; struct b *i; } y;
struct c { struct c *i; int h; } z;
```

The first two have the same components in the same order, but the field names are different. The second and third have the same field names naming the same components, but the order of those components is different. Depending on the rules of the language, either pair could be equivalent or all three could be distinct.

A designer specifies possibly-equivalent types by partitioning a subset of the set of types such that all of the types in a particular block of the partition *might* be equivalent according to the rules of the language. Types assigned to different blocks can never be equivalent. If a type is not assigned to any block, then it is assumed to be unique. An ordered (possibly empty) set of components may be associated with each type when it is assigned to a block.

Let '`type`' and '`set`' be definition table keys, and '`components`' be a list of definition table keys. `AddTypeToBlock('type','block','components')` adds type '`type`' to the partition block defined by '`block`'. It also sets the `DefTableKeyList`-valued property `ComponentTypes` of '`type`' to '`components`'.

Suppose that the designer chose to assign every structure type to the same set (represented by a known key), and to list the field types in order of appearance. Then variables `x` and `y` above would have the same type, but `z` would have a different type. Another possibility would be to generate a unique definition table key on the basis of the sorted list of field identifiers, and then to list the field types in the order of their sorted identifiers. Variables `y` and `z` would then have the same type and `x` would have a different type.

5.2 Computing equivalence classes

Let '`S1`',...,'`Sp`' be the partition established by invocations of `AddTypeToBlock`. For each type '`t`', let '`f1(t)`',...,'`fn(t)`' be the ordered list of the component types.

Computations supplied by the `StructEquiv` module then find the partition {'`E1`',...,'`Eq`'} having fewest blocks '`Ei`' such that:

1. Each '`Ei`' is a subset of some '`Sj`'.
2. '`x`' and '`y`' in '`Ei`' implies that '`fj(x)`' and '`fj(y)`' are in some one '`Ek`', for all '`fj`'.

The blocks ‘Ei’ are the equivalence classes determined by refining the original partition introduced by AddTypeToBlock on the basis of the component types.

The algorithm then selects an arbitrary member of each ‘Ei’ as the representative type for that equivalence class, and alters the properties of the other members of that class so that they act as type identifiers pointing to the key for the representative type (see Section 7.1 [Dependences among types and type identifiers], page 40). This means that the values of an arbitrary property of the key used to represent a type in subsequent computation may not be the value of that property set at a specific instance of a type denotation for that type (see Section 4.3 [Referring to a type], page 25).

5.3 Functions as typed entities

Many languages have the concept that a function is a typed entity. Such a language provides a form of type denotation that can describe function types. Function definitions also implicitly describe function types, since there is usually no way of using a type identifier to specify the type of a function. Thus every function definition must also be considered a type denotation.

Function definitions are operator definitions, defining an operator that is used verify the type-correctness of the function invocation. Because the structural equivalence algorithm will select an arbitrary element to represent the equivalence class, every function type denotation must also define an invoker.

Modula-3 has constructs representing type denotations for function types (ProcTy) and function definitions (Procedure) that could be specified as follows (a function invocation is also given):

```
SYMBOL Formals INHERITS OpndTypeListRoot             END;
SYMBOL ProcTy  INHERITS TypeDenotation, OperatorDefs END;

RULE: ProcTy ::= 'PROCEDURE' '(' Formals ')' ':' Type COMPUTE
  .Indic=NewKey();
  ProcTy.GotType+=
    ORDER(
      ResetInvoker(ProcTy.Type,.Indic),
      AddTypeToBlock(
        ProcTy.Type,
        procClass,
        ConsDefTableKeyList(Type.Type,Formals.ParameterTypeList)));
  ProcTy.GotOper+=
    ListOperator(.Indic,NoOprName,Formals.ParameterTypeList,Type.Type);
END;
```

```
SYMBOL Procedure INHERITS TypeDenotation, OperatorDefs END;

RULE: Procedure ::= '(' Formals ')' ':' Type '=' Block COMPUTE
  Procedure.EqClass=procClass;
  Procedure.ComponentTypes=
    ConsDefTableKeyList(Type.Type,Formals.ParameterTypeList);
  .Indic=NewKey();
  Procedure.GotType+=
    ORDER(
      ResetInvoker(Procedure.Type,.Indic),
      AddTypeToBlock(
        Procedure.Type,
        procClass,
        ConsDefTableKeyList(Type.Type,Formals.ParameterTypeList)));
  Procedure.GotOper+=
    ListOperator(.Indic,NoOprName,Formals.ParameterTypeList,Type.Type);
END;


SYMBOL Expr    INHERITS ExpressionSymbol END;
SYMBOL Actuals INHERITS OpndExprListRoot END;

RULE: Expr ::= Expr '(' Actuals ')' COMPUTE
  ListContext(Expr[1],,Actuals);
  Indication(GetInvoker(Expr[2].Type,NoKey));
END;
```

6 Error Reporting in Type Analysis

Language-dependent error reporting involves checks based on the types associated with program constructs by the computations specified in earlier chapters. For example, object-oriented languages differ in their requirements for overriding methods when extending a class definition. One possibility is to require that the type of each parameter of the overriding method be a supertype of the corresponding parameter type of the overridden method, and that the result type of the overriding method be a subtype of the result type of the overridden method. The type analysis modules will establish the complete signatures of both methods, and the subtype/supertype relation among all type pairs. Thus only the actual check remains to be written.

Some errors make it impossible to associate any type with a program construct, and these are reported by the modules. Operations are also made available to support detection of incorrect typing.

6.1 Verifying typed identifier usage

An applied occurrence of an identifier that purports to represent a typed entity inherits the `TypedIdUse` role. The value of its `Type` attribute should not be `NoKey`, and the identifier itself should not be a type identifier. Both of these conditions can be checked by inheriting the `ChkTypedUseId` role:

```
SYMBOL ExpIdUse INHERITS ChkTypedUseId END;
```

If the identifier '`id`' at an `ExpIdUse` node is bound, but the type is unknown, the `ChkTypedUseId` computation will issue the following report at the source coordinates of '`id`'

```
Must denote a typed object: 'id'
```

If the identifier '`id`' at an `ExpIdUse` node is a type identifier, the report would be:

```
Type identifier not allowed: 'id'
```

6.2 Verifying type identifier usage

Both defining and applied occurrences of type identifiers can be checked for validity. In each case, the value of the `Type` attribute must be a definition table key whose `IsType` property has the value 1. Two roles are available for this purpose:

`ChkTypeDefDefId`
: reports an error if the `Type` attribute does not refer to a type, or if the type refers to itself.

`ChkTypeDefUseId`
: reports an error if the `Type` attribute does not refer to a type.

6.3 Verifying type consistency within an expression

The `Expression` module provides default error reporting associated with the following roles:

`ExpressionSymbol`
: Condition: `'e'.Type` is not acceptable as `'e'.Required`.

Message: 'Incorrect type for this context'

Override symbols:
ExpMsg, ExpErr, ExpError

OperatorSymbol

Condition: The indication is valid but no operator could be identified.

Message: 'Incorrect operand type(s) for this operator'

Override symbols:
OprMsg, OprErr, OprError

OpndExprListRoot

Condition: The function requires more arguments than are present.

Message: 'Too few arguments'

Override symbols:
LstMsg, LstErr, LstError

OpndExprListElem

Condition: The function requires fewer arguments than are present.

Message: 'Too many arguments'

Override symbols:
ArgMsg, ArgErr, ArgError

This error reporting can be changed by overriding computations for the 'xxx'Msg attribute. The 'xxx'Err attribute has the value 1 if the error condition is met, 0 otherwise. Thus the overriding computation might be of the form:

```
's'.'xxx'Msg=
  IF('s'.'xxx'Err,message(ERROR,"My report",0,COORDREF));
```

Because 's'.'xxx'Msg is of type VOID, you can remove a report completely by setting 's'.'xxx'Msg to "no".

If you wish to override the message in every context, write the overriding computation as a symbol computation in the lower context of the override symbol specified above. In this case, 'xxx' would be SYNT. Here is an example, changing the error report for invalid operators in all contexts:

```
SYMBOL OprError COMPUTE
  SYNT.OprMsg=
    IF(SYNT.OprErr,message(ERROR,"Invalid operator",0,COORDREF));
END;
```

If you wish to override the message in a few specific contexts, write the overriding computation as a rule computation in the lower context of a symbol inheriting the computational role. In this case, 'xxx' would be the symbol on the left-hand side of the rule. Here is an example, changing the standard expression error report to be more specific for function arguments:

```
RULE: Actual ::= Expr COMPUTE
  Actual.ExpMsg=
    IF(Actual.ExpErr,message(ERROR,"Wrong argument type",0,COORDREF));
END;
```

6.4 Support for context checking

As noted in the previous section, `OperatorSymbol` role computations normally report an error when an indication is valid but no operator can be identified. The `Expression` module exports two context-dependent rule computations for use when an expression node has no children playing that role. One computation tests the indication and the other tests the operator:

`BadIndication`
: Yields 1 if the operator indication supplied by `Indication` is unknown, 0 otherwise.

`BadOperator`
: Yields 1 if the indication is valid but no operator can be selected from that indication's set, 0 otherwise.

Consider an expression in which a function is applied to arguments (see Section 5.3 [Functions as typed entities], page 32):

```
SYMBOL Expr    INHERITS ExpressionSymbol END;
SYMBOL Actuals INHERITS OpndExprListRoot END;

RULE: Expr ::= Expr '(' Actuals ')' COMPUTE
  ListContext(Expr[1],,Actuals);
  Indication(GetInvoker(Expr[2].Type,NoKey));
  IF(BadIndication,
    message(ERROR,"Invalid function",0,COORDREF));
END;
```

Suppose that, because of a programming error, `Expr[2]` does not deliver a function type. In that case, `Expr[2].Type` would not have the `Invoker` property, and `BadIndication` would yield 1. Alternatively, `Expr[2]` might deliver a function whose signature does not match the context. Because the indication has only a singleton operator set, that operator will be selected regardless of the context. Errors will then be reported by the default mechanisms as an incorrect number of arguments, arguments of incorrect types, or result incorrect for the context.

Now consider the array access expression (see Section 3.4 [Operators with explicit operands], page 17):

```
SYMBOL Subscript INHERITS ExpressionSymbol END;

RULE: Expr ::= Expr '[' Subscript ']' COMPUTE
  DyadicContext(Expr[1],,Expr[2],Subscript);
  Indication(indexInd),
  IF(BadOperator,
    message(ERROR,"Invalid array reference",0,COORDREF));
END;
```

Suppose that, because of a programming error, `Expr[2]` does not deliver an array type. In that case, there would be no operator in `indexInd`'s operator set whose left operand was the type returned by `Expr[2]` and `BadOperator` would yield 1.

It is sometimes useful to be able to check whether one type is acceptable as another outside of the situations covered in the previous section. Let '`from`' and '`to`' be definition table keys representing types. `IsCoercible('from','to')` yields 1 if a value of type '`from`' is acceptable wherever an value of type '`to`' is required; it yields 0 otherwise.

For example, consider a cast involving a reference type in Java. The cast is known to be correct at compile time if a value is being cast to its superclass. If the value is being cast to one of its subclasses, however, a run-time check is required. Thus the compiler must accept such a cast *both* when the value is acceptable as a value of the cast type *and* when a value of the cast type is acceptable as a value of the type being cast:

```
RULE: Expression ::= '(' Expression ')' Expression COMPUTE
  IF(AND(
      NOT(IsCoercible(Expression[2].Type,Expression[3].Type)),
      NOT(IsCoercible(Expression[3].Type,Expression[2].Type))),
    message(ERROR,"Invalid cast",0,COORDREF));
END;
```

7 Dependence in Type Analysis

Type analysis is a complex process, involving several different kinds of entity. Each kind of entity has properties, which are stored in the definition table under the entity's key. Those properties are set and used in a variety of contexts. The result is a collection of implicit dependence relations among the type analysis computations, and these relations depend on the language being analyzed.

The modules described in this document make the implicit relations explicit, using void attributes and dependent expressions in LIDO (see Section "Dependent Expressions" in *LIDO - Reference Manual*). Although the explicit dependences work for a wide range of typical programming languages, one or more of them must sometimes be overridden because of the rules of a particular language. This chapter explains the implicit dependences that must be made explicit, how the various modules make them explicit, and some typical circumstances in which the default treatment fails.

The void attributes that make these dependences explicit are summarized here; the remainder of this chapter explains them in more detail:

`TypeDenotation.GotType`
: The new type key has been created, and any properties that are not dependent on final types have been stored in the definition table as properties of that key.

`TypeDefDefId.GotDefer`
: Information that can be used to find the final type has been stored in the definition table as properties of the key assigned to the identifier by the name analyzer.

`RootType.GotUserTypes`
: Computations for all type denotations have reached the state represented by `TypeDenotation.GotType` and computations for all type identifier definitions have reached the state represented by `TypeDefDefId.GotDefer`.

`RootType.GotAllTypes`
: All final types have been determined.

`TypedDefId.TypeIsSet`
: Information that can be used to find the final type has been stored in the definition table as properties of the key assigned to the identifier by the name analyzer.

`RootType.TypeIsSet`
: The state represented by `RootType.GotAllTypes` has been reached, and computations for all typed identifier definitions have reached the state represented by `TypedDefId.TypeIsSet`.

`TypedUseId.TypeIsSet`
: All information needed to find the final type of this typed identifier is available.

`OperatorDefs.GotOper`
: All operator descriptions associated with this construct have been entered into the operator data base.

`RootType.GotAllOpers`
: Computations for all symbols inheriting `OperatorDefs` have reached the state represented by `OperatorDefs.GotOper`.

7.1 Dependences among types and type identifiers

Consider the following program, written in a C-like notation:

```
{ Measurement Length;
  typedef Inches Measurement;
  typedef int Inches;

  Length = 3; printf("%d\n", Length + 7);
}
```

Suppose that the language definition states that type identifiers are statically bound, with the scope of a declaration being the entire block. Thus all of the type identifier occurrences have valid bindings. (That would *not* be the case in C, because in C the scope of a declaration is from the end of the declaration to the end of the block.)

The type analysis of each of the two occurrences of `Length` in the last line of the program is described by the following specifications discussed earlier:

```
SYMBOL ExpIdUse INHERITS TypedUseId, ChkTypedUseId END;

RULE: Expr ::= ExpIdUse COMPUTE
  PrimaryContext(Expr,ExpIdUse.Type);
END;
```

The value of `ExpIdUse.Type` should be `intType`, the known definition table key created for the language-defined integer type. Recall that `intType` was associated with the `int` keyword by the following specification:

```
RULE: Type ::= 'int' COMPUTE
  Type.Type=intType;
END;
```

The problem is to make the `intType` value of the `Type.Type` attribute in this context the value of the `ExpIdUse.Type` attribute in the context quoted above.

A human has no trouble seeing how this problem could be solved:

1. The type definition rule sets the `TypIdDef.Type` attribute of the occurrence of `Inches` in the third line of the program to `intType`.
2. The value of a property of the `Inches` entity could be set from the value of the `TypIdDef.Type` attribute in that context.
3. That property could be used to set the `TypIdUse.Type` attribute of the occurrence of `Inches` in the second line of the program.
4. The type identifier use rule sets the value of the `Type.Type` attribute in that context the value of the `TypIdUse.Type` attribute.
5. Similar reasoning results in the value of the `Type.Type` attribute in the variable definition context of the first line of the program becoming `intType`.

6. Finally, a property of the `Length` entity is set in the context of the first line of the program and used to make `intType` the value of the `ExpIdUse.Type` attributes in the two contexts of the last line.

Unfortunately, this solution is based on the human's ability to see the dependence among the type identifiers and process the lines of the program in an order determined by that dependence. One cannot, for example, blindly process the lines in the order in which they were written.

The dependence among the lines in our example is a result of our use of the known key `intType` as the value of a property of the type identifier entities. This strategy is actually an example of a premature evaluation: There is no need to know the key representing the type of `Length` until the `ExpIdUse.Type` attribute is evaluated. We can avoid the constraint on the order of rule processing by a "lazy evaluation" strategy in which we use properties of the type identifier entities to establish a means for determining the value of the `ExpIdUse.Type` attribute rather than establishing the value itself.

Recall that there are three possible `Type` contexts: a keyword, a type denotation, and a type identifier (see Section 4.3 [Referring to a type], page 25). In the first two, we can set the value of the `Type.Type` attribute to the definition table key for the type itself. In the third, however, the only information that we are guaranteed to have is the definition table key for the type identifier. However, this information is sufficient to find the definition table key for the type *once all of the type identifiers have been defined*. Thus we can simply set the value of the `Type.Type` attribute to the definition table key for the type identifier itself in this context.

The computation provided by the `Typing` module for the `TypeDefDefId` context sets a property of the type identifier entity to the value of the `TypeDefDefId.Type` attribute (see Section 4.2 [Type identifiers], page 24). Effectively, this computation creates a linear list of type identifier entities ending in a type entity. When all of the entities corresponding to type identifiers have this property set, the definition table key for a type should be the last element of each list.

In our example, the value of this property of the identifier `Length`'s definition table key would be the definition table key of the identifier `Measurement`. The value of its property would be the definition table key of the identifier `Inches`, whose property would be `intType`.

There is no guarantee, of course, that the last element of the list is actually a type. For example, consider the following incorrect program:

```
{ Measurement Length;
  typedef Inches Measurement;
  int Inches;

  Length = 3; printf("%d\n", Length + 7);
}
```

Here the last element of the list beginning at `Measurement` would be `Inches`, a variable identifier. The `ChkTypeDefUseId` role checks the `IsType` property of the key that is the last element of the list to report errors of this kind (see Section 6.2 [Verifying type identifier usage], page 35).

The void attribute `RootType.GotUserTypes` represents the state of the computation at which all of the type denotations and type identifiers have been formed into lists.

7.2 Dependence on structural equivalence

The structural equivalence computation must be carried out after all `Type.Type` attributes have been set and linked as described in the previous section, and all of the possibly-equivalent types have been added to the appropriate blocks of the initial partition (see Section 5.2 [Computing equivalence classes], page 31). The latter condition is represented by all of the void attributes `TypeDenotation.GotType` having been set. A user can override the computation of the void attribute `RootType.GotType` to signal dependence of the structural equivalence computation on any additional information.

`RootType.GotAllTypes` is the post-condition for the structural equivalence algorithm. After that computation is complete, however, some definition table keys that were thought to represent types have had their properties changed so that they represent type identifiers (see Chapter 5 [Structural Type Equivalence], page 31). Thus scanning a list of definition table keys to find the last one is only meaningful after `RootType.GotAllTypes` has been established. The `TypeKey` attributes of `TypeDenotation`, `TypeDefDefId`, and `TypeDefUseId` reflect this fact.

Sometimes a designer uses a C routine to access type properties. If the keys defining the types have been obtained from `Type` attributes of `TypeDenotation`, `TypeDefDefId`, or `TypeDefUseId` (rather than from `TypeKey` attributes of those nodes), then `FinalType` can be used to obtain the key at the end of the list. The C program must include the header file `Typing.h`, and the code must enforce a dependence on `RootType.GotAllTypes`. If that dependence is not enforced, the results of invoking `FinalType` are undefined.

7.3 Dependence on the operator database

The operator identification database used for type analysis within expressions is initialized from the specifications of language-defined types, operators, and operator indications. Database representations for function operators and operators associated with user-defined types cannot be constructed until the pre-condition `RootType.GotAllTypes` has been established. Moreover, type analysis of expressions cannot be carried out until that information has been entered into the database.

Computations that define operators must establish the `GotOper` post-condition at their associated `OperatorDefs` nodes. The computation of `RootType.GotOper` can be overridden to provide dependence on computations not associated with an `OperatorDefs` node. `RootType.GotAllOpers` represents the state in which the database has been completely populated. All expression analysis computations have `RootType.GotAllOpers` as a pre-condition.

7.4 Dependences for typed entities

The computation provided for the `TypedDefId` context sets the `TypeOf` property of that identifier's key to the value of `TypedDefId.Type`. `TypedDefId.TypeIsSet` is the post-condition for that computation. `RootType.AllTypesAreSet` is the conjunction of all of the `TypedDefId.TypeIsSet` post-conditions plus `RootType.GotAllTypes`.

If other property values of the identifier's key are set by user computations in the lower context of `TypedDefId` that establish the postcondition `SYNT.GotProp`, then the setting of these properties is also guaranteed by the post-condition `TypedDefId.TypeIsSet`.

(SYNT.GotProp defaults to the empty postcondition.) Note that if any of these user computations depend on any results from type analysis, a cycle will be created.

A computation supplied by the module sets the TypedUseId.Type attribute to the value of the TypeOf property of that identifier's definition table key. TypedUseId.TypeIsSet is a precondition for that computation. It must guarantee that the TypeOf property of the identifier has actually been set. The module provides a default computation for TypedUseId.TypeIsSet in the lower context of the TypedUseId node, requiring the precondition RootType.TypeIsSet.

Some languages provide initialized variable declarations, and allow the user to omit either the type specification or the initializing expression but not both. If the type specification is omitted, the variable's type is the type returned by the initializing expression. Here are some examples of such declarations in Modula-3:

```
VAR Both: INTEGER := 3;
VAR NoType := Both + 7;
VAR NoInit: INTEGER;
```

The default computations for the TypeIsSet attributes in this example lead to a cycle:

1. The TypedDefId.Type attribute of NoType depends on TypedUseId.Type for Both.
2. The computation of TypedUseId.Type for Both has the pre-condition TypedUseId.TypeIsSet.
3. TypedUseId.TypeIsSet depends on RootType.TypeIsSet in the default computation.
4. RootType.TypeIsSet is the conjunction of *all* TypedDefId.TypeIsSet attributes.
5. TypedDefId.TypeIsSet for NoType is the post-condition for a computation involving the TypedDefId.Type attribute of NoType.

If the language requires that the initialized declaration of a variable precede any uses of that variable, then we can override the default dependence as follows:

```
CHAIN TypeDepend: VOID;

CLASS SYMBOL ROOTCLASS COMPUTE
  CHAINSTART HEAD.TypeDepend=THIS.GotType;
END;

RULE: VrblDecl ::= 'VAR' VarIdDef ':' Type ':=' Expr COMPUTE
  VrblDecl.Type=Type.Type;
  VrblDecl.TypeDepend=VarIdDef.TypeIsSet <- Expr.TypeDepend;
END;

RULE: VrblDecl ::= 'VAR' VarIdDef ':' Type COMPUTE
  VrblDecl.Type=Type.Type;
  VrblDecl.TypeDepend=VarIdDef.TypeIsSet <- Expr.TypeDepend;
END;

RULE: VrblDecl ::= 'VAR' VarIdDef ':=' Expr COMPUTE
  VrblDecl.Type=Expr.Type;
  VrblDecl.TypeDepend=VarIdDef.TypeIsSet <- Expr.TypeDepend;
END;
```

```
SYMBOL VarIdUse COMPUTE
  SYNT.TypeIsSet=THIS.TypeDepend;
  THIS.TypeDepend=SYNT.TypeIsSet;
END;
```

If there is no ordering requirement, then a fixed-point computation is required to determine the variable types. In addition, code generated from the initializing expressions must be arranged to ensure that a variable's value is computed before it is used. Finally, such out-of-order dependence makes the program hard to understand. We strongly recommend that declaration before use be required if variables are allowed to obtain their type from their initializers.

8 Program-Dependent Property Computation

There are language constructs which need to set a property of an entity that depends on a property of another entity. If the order of accessing and setting the properties depends on the particular program, those operations can not be specified by dependent computations in trees, rather a program dependent mechanism is needed. An example for such a construct is a variable declaration of the form

```
like a b;
```

The variable `b` is declared to have the same type as the variable `a` has. A program may have arbitrary long chains of such type references, which may occur in any order. This module provides a worklist algorithm to solve such problems: Tree computations create worklist tasks, which try to access and set certain properties as described by call-back functions, and store them on the worklist. They are re-executed in sweeps through the worklist until no futher task can be completed.

The module is instantiated by

```
$/Type/PropertyWorklist.gnrc+instance=‘prefix’:inst
```

If the `instance` parameter is given, the `prefix` is added to every name exported by that instantiation, e.g. `prefixTaskFct`. In the following we write `*TaskFct` to indicate that such a prefix will be added to that name. Several instances of the module with different prefixes may be used, if several worklists are needed,

The module provides the roles `*RootWorklist` and `*WLPropertyTask`:

`*WLPropertyTask` is the central task which creates a worklist task. The attribute `SYNT.*TaskFct` has to be set to a pointer to a function which performs the computation of the worklist task for this context. (Different functions may be chosen in different `*WLPropertyTask` contexts.) The values of the following attributes are available when the function is called during sweeps through the worklist:

`INH.*PropKeyA` and `SYNT.*PropKeyB` are keys of type `DefTableKey` and have the default `NoKey`. `SYNT.*WLDepOnTaskA` and `SYNT.*WLDepOnTaskB` are pointers of type `PropertyTaskPtr` which may be set to point to other worklist task; the default is `NULLPropertyTaskPtr`.

The attribute `SYNT.*WLTask` provides a pointer which identifies the created worklist task. Its value may be propagated to some other `*WLPropertyTask` context, the task of which depends on information from this task.

Each function used as a `*TaskFct` in a `*WLPropertyTask` context has to be declared in a C module where `PropertyWL.h` is included in the `.h` and `.c` file.

All these function declarations have the following signature:

```
void WLCallBackSetType (PropertyTaskPtr this)
```

In the body of the function the values of the four attributes, e.g. `INH.*PropKeyA`, can be accessed via the parameter, e.g. `this->PropKeyA`. Furthermore, the coordinates of the creation context are available by `this->coord` of type `CoordPtr`.

In case of the above example with the `like` construct one could use `this->PropKeyA` for the key of the variable, the type of which is to be set, and `this->PropKeyB` for the variable which provides the type. The worklist algorithm while sweeping through the list of tasks re-calls the function for each task until its `this->done` is set. In this example, the function would execute

```
DefTableKey tp = GetTypeOf (this->PropKeyB, NoKey);
if (tp == NoKey) return;
ResetTypeOf (this->PropKeyA, tp);
this->done = 1; return;
```

Between the start of the worklist algorithm and its completion, no computations in tree contexts are executed. Hence, there are only two ways to propagate information from one worklist task to another: either via setting and accessing properties of `this->PropKeyA` or `this->PropKeyB`, or by accessing the data components of another task via `this->WLDepOnTaskA` or `this->WLDepOnTaskB`, e.g. `GetTypeOf (this->WLDepOnTaskA->PropKeyA, NoKey)`.

This feature can be used to substitute the attribute value propagation between related `*WLPropertyTask` contexts. Binding of qualified names like `a.b.c` is a typical example: Assume that the `TypeOf` property of variable identifiers is determined by worklist tasks, e.g. because of the presence of `like` constructs. Then the task for binding the qualified identifier `c` depends on the result of the task for the qualifier `a.b`. That relation can be established in the task creation context using the attribute `*WLDepOnTaskA`, such that the function call for the worklist task can access the `TypeOf` property.

The attribute `*RootWorklist.*WLSolved` indicates that the worklist algorithm has terminated. Any computation which accesses properties computed by worklist tasks have to depend on it. The worklist algorithm terminates, when a sweep through the list did not complete any task. Some tasks may still be unsolved, e.g. because of missing settings of properties or cyclic dependences between tasks. The attribute `*RootWorklist.*WLOpenTasks` gives the number tasks that remained unsolved. In the creation context of a task it can be checked whether this particular task completed, using

```
PTRSELECT (THIS.*WLTask, done) <- INCLUDING *RootWorklist.*WLSolved
```

The accumulating attribute `*RootWorklist.*WLReadyToSolve` contributes to the precondition for starting the worklist algorithm. It has to be used if some worklist tasks are created without notification by the `*WLPropertyTask` role. The function which creates a worklist task has the following signature:

```
PropertyTaskPtr CreatePropertyTask
        (PropWLPtr wl,
         DefTableKey PropKeyA, DefTableKey PropKeyB,
 PropertyTaskPtr WLDepOnTaskA, PropertyTaskPtr WLDepOnTaskB,
 WLTaskFctType TaskFct,
         CoordPtr coord);
```

The first parameter is the reference to the list of worklist tasks, it is obtained from the attribute `INCLUDING *RootWorklist.*Worklist`; the meaning of the other parameters can be deduced from their names. It is possible and may be useful in certain cases to call `CreatePropertyTask` from inside a worklist function; then care must be taken to guarantee termination. If the `CreatePropertyTask` is called in a C module, `PropertyWL.h` is to be included.

Index

Symbol roles

Rule computations

O

P

R

T

Attributes

G

I

O

P

R

T

General concepts

C

E

I

N

O

P

S

T