Tutorial on Type Analysis

$Revision: 3.14 $

Compiler and Programming Language Group
University of Paderborn, FB 17
33098 Paderborn, FRG

Table of Contents

Overview

This tutorial is an introduction to the solution of the type analysis required for programming language implementation. It demonstrates many aspects of that task which can occur in programming languages or special purpose languages. The solution proceeds from basic aspects, such as declarations, typed objects, and overloaded operators, up to more complex aspects, such as type definitions and function types.

This tutorial may be used for a practical introduction to the specification techniques for type analysis, or as a source of examples that show how to solve certain problems, or it may give hints for language design.

This file is an executable specification. An analyzer for an artificial language can be generated from it. The language is kept small by restricting it to those constructs necessary to demonstrate the type analysis task, not regarding its usability for programming. The generated analyzer produces output that reports the result of type analysis, i. e. the type property of program objects.

The explanations in this tutorial assume that the reader is familiar with the use of the Eli system, with the use of its module library, and knows the general description of the type analysis task in the documentation of the module library. Furthermore, the reader should be familiar with basic concepts of the name analysis task. Its solution is a precondition for type analysis. It is described only briefly in this text. There is a separate tutorial for name analysis, see Section 'Overview' in Tutorial on Name Analysis.

Sections 1 to 4 specify a small language kernel and solve the name analysis task. The focus of the following sections is on type analysis only. The topics are arranged in an order such that no forward references are needed. Readers may stop reading at any section after section 7. They then have a complete description of the name analysis task for a language that has the constructs and concepts introduced so far. (It should even be possible to drop the rest of the specification an generate an analyzer for the language specified so far. This feature has not yet been tried.) For that purpose the example language is presented such that from section 5 on each section augments the language by some new constructs and concepts that demonstrate the aspect of concern. Readers are asked for some patience until they see what the whole language is.

1 Kernel Language

We start with a very simple kernel language where a `Program` is a `Block` consisting of `Declarations` for variables, assignment `Statements`, and trivial `Expressions`. Other forms of `Declarations` and `Expressions` are added to the grammar when the type analysis task is further refined.

Here is a simple example program: **SimpleExamp**[1]==

```
begin
  var   int i, int j,
        bool b, bool c,
        real r, real s;
  i = 1;
  b = true;
  r = 3.4;
  j = i;
  c = b;
  s = r;
end
```

This macro is attached to a product file.

The concrete kernel grammar is specified as follows: **Core.con**[2]==

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

Declaration:    'var' ObjDecls ';'.
ObjDecls:       ObjDecl // ',' / .
ObjDecl:        TypeDenoter DefIdent.
TypeDenoter:    TypeUseIdent.

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

Expression:     Factor.
Factor:         Operand.
Operand:        IntNumber.
Operand:        RealNumber.
Operand:        Variable.
Variable:       UseIdent.
```

This macro is attached to a product file.

The expression syntax is prepared to introduce operators of different precedences (2 for binary and 1 for unary operators). `Factor` and `Operand` are represented by `Expression` contexts in the tree grammar.

Opr.sym[3]==

```
Expression ::= Factor Operand.
```

This macro is attached to a product file.

The notation of identifiers and numbers is chosen as in Pascal.

Core.gla[4]==

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

This macro is attached to a product file.

2 Basic Scope Rules

The basic task of name analysis is consistent renaming. For each identifier occurrence a `Key` attribute is computed such that identifier occurrences that refer to the same object in the program have the same `Key` attribute value. Hence `Key`s identify program objects uniquely. `Key`s are used to associate properties to program objects and to retrieve those properties in different contexts.

For the kernel language we distinguish the following identifier occurrences:

Scope.con[5]==

```
DefIdent:        Ident.
UseIdent:        Ident.
TypeUseIdent:    Ident.
```

This macro is attached to a product file.

The scope rules of a language determine how identifier occurrences are bound to program objects. We specify Algol-like scope rules for our language. The basic Algol-like scope rule reads:

> A definition of an identifier `a` is valid in the whole smallest range that encloses that definition, except inner ranges that contain another definition of `a`.

Hence, a definition in an outer range is hidden by a definition of the same identifier in an inner range for the whole inner range. Identifiers may be applied before they are defined.

We instantiate a library module that provides computations according to this scope rule:

CoreScope.specs[6]==

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

This macro is attached to a product file.

The use of that module requires that every identifier occurrence has the attribute `Sym` representing the identifier encoding. Hence we specify a computational role `IdentOcc` that provides that attribute, and will be inherited to any identifier occurrence.

The computational roles `RangeScope`, `IdDefScope`, and `IdUseEnv` are associated to the corresponding symbols of our grammar.

CoreScope.lido[7]==

```
TERM Ident: int;
ATTR Sym: int;
CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;

SYMBOL Block    INHERITS RangeScope END;
SYMBOL DefIdent INHERITS IdDefScope, IdentOcc END;
SYMBOL UseIdent INHERITS IdUseEnv, IdentOcc END;
SYMBOL TypeUseIdent INHERITS IdUseEnv, IdentOcc END;
```

This macro is attached to a product file.

3 Messages on Scope Rule Violations

Erroneous programs may violate the scope rules in one of two cases:

- A particular applied identifier occurrence has no valid defining identifier occurrence.
- There are more than one defining identifier occurrences for one identifier in one range.

Such situations shall be indicated by error messages. Furthermore, we want every defining occurrence of a multiply defined identifier be marked by a message.

For that purpose we use the following two library modules:

CoreChk.specs[8]==

```
$/Tech/Strings.specs
$/Prop/OccCnt.gnrc:inst
```

This macro is attached to a product file.

The `Strings` module provides a function that concatenates a error message string and an identifier for error messages related to identifiers.

The `OccCnt` module provides computations that count how often an object identified by a `Key` attribute occurs in certain contexts, in our case in a defining context.

The check for existence of a definition is directly obtained from the module role `ChkIdUse`. For the second check we specify a computational role `ChkUnique` in order to reuse it for several grammar symbols. If an object occurs more than once in the `ChkUnique` context it is multiply defined.

CoreChk.lido[9]==

```
SYMBOL UseIdent INHERITS ChkIdUse END;
SYMBOL TypeUseIdent INHERITS ChkIdUse END;
SYMBOL DefIdent INHERITS ChkUnique END;

SYMBOL ChkUnique INHERITS Count, TotalCnt COMPUTE
  IF (GT (THIS.TotalCnt, 1),
  message (ERROR,
```

```
            CatStrInd ("identifier is multiply defined: ",
                        THIS.Sym),
            0, COORDREF));
END;
```

This macro is attached to a product file.

4 Properties of Predefined Objects

In our language the type names `int`, `real`, `bool`, and `void`, and the names `true` and `false` for boolean values are predefined. They are introduced using the following library modules:

Predef.specs[10]==

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

This macro is attached to a product file.

The following description defines a key for each of the predefined identifiers:

Predef.d[11]==

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

This macro is attached to a product file.

As we are going to use the library module `Typing` to support type analysis we have to adopt its strategy for representing types:

Types are represented by `DefTableKeys`. Such a key is created for each program construct which denotes a particular type, and for each predefined type, i. e. the keys `intType`, etc. below. The property `IsType` distinguishes them from non-type objects. These type keys have to be different from the keys associated to the predefined type names, `intKey`, etc. The former are associated to the latter via the property `Defer`. This technique allows the module roles to also deal with type definitions, and allows different type names stand for the same type.

PredefTypes.pdl[12]==

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

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

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

This macro is attached to a product file.

Typed objects, like variables, have the property `TypeOf` that represents their type. Hence, we associate that property with the predefined names for boolean values:

PredefConsts.pdl[13]==

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

This macro is attached to a product file.

Usually the properties `TypeOf`, `Defer`, and `IsType` are not set or accessed explicitly. Module roles ensure that they are properly used.

5 Declaration of Typed Objects

We here consider a variable declaration as an example for a language construct that defines a typed object. In our language a variable declaration may define several variables. An `ObjDecl` states the type and the name for each of them. Until now we allow types to be denoted only by predefined type identifiers, `TypeUseIdent`. Further forms of `TypeDenoter` are specified below.

We use the module `Typing` for support of type analysis.

Declare.specs[14]==

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

This macro is attached to a product file.

Types are represented by `DefTableKeys`. Such a key is created for each program construct which denotes a particular type. The unknown type is represented by `NoKey`. Typed objects, like variables, have the property `TypeOf` that represents their type.

The pair of module roles `TypedDefinition` and `TypedDefId` support the pattern of declaring typed objects: `ObjDecl` has the role `TypedDefinition`, i. e. a construct that specifies the types of all `TypedDefIds` in its subtree. The attribute `ObjDecl.Type` has to be set appropriately.

Declare.lido[15]==

```
SYMBOL ObjDecl INHERITS TypedDefinition END;
SYMBOL DefIdent INHERITS TypedDefId END;

ATTR Type: DefTableKey;

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

This macro is attached to a product file.

The module roles `TypedUseId` and `TypeDefUseId` distinguish between used names of typed objects and used names of types. The corresponding check roles issue messages if that classification is violated.

The use of `TypedUseId` causes the attribute `UseIdent.Type` to be set to the type defined for the object.

TypeUseIdent.lido[16]==

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

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

This macro is attached to a product file.

In order to report some results of the type analysis we associate the string valued property `TypeName` to every type key.

The specification `[KReset]` causes an additional definition table operation to be created, such that a call `KResetTypeName (k, v)` sets the value `v` to the key `k` and returns that key. That allows us to nest several calls of functions which set properties to one key. The definition of `KReset` is obtained from the `PropLib` module which is available when using the `Typing` module.

PrtType.pdl[17]==

```
TypeName: CharPtr [KReset]; "Strings.h"

intType ->  TypeName = {"int"};
realType -> TypeName = {"real"};
boolType -> TypeName = {"bool"};
voidType -> TypeName = {"void"};
```

This macro is attached to a product file.

For every used identifier that string is to be printed:

PrtType.lido[18]==

```
SYMBOL UseIdent INHERITS PrtType END;

SYMBOL PrtType COMPUTE
  printf ("line %d Type %s\n", LINE,
          GetTypeName (THIS.Type, "no type name"));
END;
```

This macro is attached to a product file.

6 Type Checking in Expressions

The type property of simple expressions is just propagated upwards using the attribute `Type`. How operators are handled is shown later.

ExprType.lido[19]==

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

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

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

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

This macro is attached to a product file.

Occurrences of `Expression` in certain contexts require that their type be compatible to a type imposed by the context. Hence, we introduce an inherited attribute `ReqType` and a check for compatibility. The function `CompatibleTypes` is defined below.

ExprTypeChk.lido[20]==

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

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

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

This macro is attached to a product file.

Compatibility of two types is specified as follows: A type `narrow` is compatible to a type `wide` if `narrow` and `wide` are equal types, or if `narrow` is the `int` type and `wide` is the `real` type, or if `wide` is the `void` type and `narrow` is any type.

Two types are equal if they stem from one occurrence of a type denoter, disregarding any renaming of that type. Further below additional rules for type equality are added.

These rule are specified by the following two functions. The function `EqualTypes` also takes care of erroneous cases in order to avoid avalanche errors: The unknown type, represented by `NoKey` is considered to be equal to any type.

Compatible.c[21]==

```
#include "Compatible.h"
#include "pdl_gen.h"
#include "eliproto.h"
#include "TypeFct.h"

#ifdef PROTO_OK
int EqualTypes (DefTableKey t1, DefTableKey t2)
#else
int EqualTypes (t1, t2) DefTableKey t1, t2;
#endif
{
  t1 = TransDefer (t1); t2 = TransDefer (t2);
  if ((t1==NoKey) || (t2==NoKey) || (t1==t2)) return 1;

/* Insertion precondition:
   t1 and t2 are different type keys, and
   both are different from NoKey.
*/
#include "EqualTypes.h"
/* Insertion postcondition:
   There is no language rule applicable that states
   t1 considered to be equal to t2.
*/
```

```
  return 0;
}

#ifdef PROTO_OK
int CompatibleTypes (DefTableKey wide, DefTableKey narrow)
#else
int CompatibleTypes (wide, narrow) DefTableKey wide, narrow;
#endif
{
  if (EqualTypes (wide, narrow)) return 1;
  wide = TransDefer (wide);
  narrow = TransDefer (narrow);
  if (wide == voidType) return 1;
  if (wide == realType && narrow == intType) return 1;
  return 0;
}
```

This macro is attached to a product file.

The implementation above includes two files `TypeFct.h` and `EqualTypes.h`. They are composed by the `phi` tool. The contributions to them are specified below, when we add further rules for type equality. The same technique is applied in the header file below.

Compatible.h[22]==

```
#include "deftbl.h"
#include "eliproto.h"

extern int EqualTypes ELI_ARG((DefTableKey t1, DefTableKey t2));
extern int CompatibleTypes ELI_ARG((DefTableKey wide, DefTableKey narrow));

#include "TypeFctHdr.h"
```

This macro is attached to a product file.

Compatible.head[23]==

```
#include "Compatible.h"
```

This macro is attached to a product file.

7 Different Kinds of Objects

After having introduced variables to our language in the last section, we need to indicate which kinds of program objects exist in the language, and how are they distinguished. So far we have types, variables, and the two predefined boolean constants being program objects. The contexts where a type is required are distinguished from other contexts by using the roles of the `Typing` module and its `IsType` property.

Furthermore we have to ensure that on the left-hand side of an assignment there is really a variable, rather than a boolean constant. (An erroneous use of a type is captured by the above mentioned distinction.) That distinction can not be made using the `TypeOf` property. We need to characterize declared variables by an additional property, and check it in the context of assignments. In our case it is sufficient just to state whether the declared object is a variable. (In more complicated languages we may need several different kinds to be distinguished.)

IsVariable.pdl[24]==

```
IsVariable: int;
```

This macro is attached to a product file.

As `ObjDecls` will be used for declaration of objects other than variables we pass an attribute `IsVariable` down from the variable declaration context the `DefIdents`, using a default value for other contexts.

The `IsVariable` property is set in the `DefIdent` context. The `GotProp` attribute is provided by the `TypedDefId` role in order to set arbitrary properties of defined objects along with the type property. The module ensures that these properties can be accessed when the type property is accessible.

IsVariable.lido[25]==

```
SYMBOL ObjDecls, DefIdent: IsVariable: int;
SYMBOL ObjDecls COMPUTE INH.IsVariable = 0; END;
SYMBOL DefIdent COMPUTE INH.IsVariable = 0; END;

RULE: Declaration ::= 'var' ObjDecls ';' COMPUTE
  ObjDecls.IsVariable = 1;
```

```
END;

RULE: ObjDecl ::= TypeDenoter DefIdent COMPUTE
  DefIdent.IsVariable = INCLUDING ObjDecls.IsVariable;
END;

SYMBOL DefIdent COMPUTE
  INH.GotProp = ResetIsVariable (THIS.Key, THIS.IsVariable);
END;
```

This macro is attached to a product file.

We now need to mark those contexts of the `Variable` symbol which are required to denote a variable object; i. e. the left-hand side of assignments, and check the `IsVariable` property there. (Other alternatives of `Variable`, which will be introduced later, can not yield a constant. Hence, we can apply the default technique for the attribute `VariableNeeded`.)

ChkVariable.lido[26]==

```
SYMBOL Variable: VariableNeeded: int;
SYMBOL Variable COMPUTE INH.VariableNeeded = 0; END;

RULE: Statement ::= Variable '=' Expression ';' COMPUTE
  Variable.VariableNeeded = 1;
END;

RULE: Variable ::= UseIdent COMPUTE
  IF (AND (Variable.VariableNeeded,
           NOT (GetIsVariable (UseIdent.Key, 0))),
  message (ERROR, "variable required", 0, COORDREF))
  <- Variable.Type;
END;
```

This macro is attached to a product file.

The dependency `<- Variable.Type` above ensures that the property `IsVariable` is not accessed before the type property of the objects is set, and the `IsVariable` along with it.

(For our simple language, where only the predefined boolean constants can violate the above condition, we could have specified the check for variables even simpler: Just state that `true` and `false` do not have the `IsVariable` property. But we wanted to demonstrate here how to set other properties along with the type property.)

8 Operator Overloading

We here extend our language by binary and unary operators in order to demonstrate type analysis for expressions with overloaded operators. The following grammar fragment introduces binary operators such that `AddOpr` has lower precedence than `MulOpr`, and unary operators, `MonOpr`, have highest precedence. (The necessary chain productions have been specified in a grammar fragment above.) The productions for the operators themselves are created from the operator descriptions below.

Opr.con[27]==

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

This macro is attached to a product file.

Type analysis for operators is supported by the library module `Operator`:

OprType.specs[28]==

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

This macro is attached to a product file.

The `referto` parameter of the instantiation identifies the following specification fragment which contains descriptions of the operators.

Operator.descr[29]==

Source Operators[30]

Target Operators[31]

This macro is attached to a product file.

Operators are overloaded in our language, i.e. a source operator symbol like `+` may denote one of several target operations, e.g. integer addition or floating point addition. The distinction is made using the types of the operands. Hence, we provide two related sets of descriptions, one for source operators and one for target operators.

Each of the following lines describes one source operator by its token, the concrete grammar symbol it is derived from, the abstract grammar symbol for its representation in the tree, and a unique name:

Source Operators[30]==

```
SrcOpr ('+', AddOpr, BinOpr, AddOp)
SrcOpr ('-', AddOpr, BinOpr, SubOp)
SrcOpr ('*', MulOpr, BinOpr, MulOp)
SrcOpr ('/', MulOpr, BinOpr, DivOp)

SrcOpr ('+', MonOpr, UnOpr, PlusOp)
SrcOpr ('-', MonOpr, UnOpr, NegOp)
SrcOpr ('!', MonOpr, UnOpr, NotOp)
```

This macro is invoked in definition 29.

For each of the source operators at least one meaning is specified by one of the following target operator descriptions. The first component of a target operator description relates it to a source operator, the second component is a unique name. Here, for example the `AddOp` is overloaded with three target operators: `iAdd`, `rAdd`, and `bAdd`. The operator names are automatically introduced as names for definition table keys. They may be used explicitly in specifications to distinguish target operators, or to associate properties to them.

The third component describes the signature of the target operator expressed in terms of keys for predefined types.

Target Operators[31]==

```
TgtOpr (AddOp, iAdd, (intType,intType):intType, TgtStr={"+"})
TgtOpr (SubOp, iSub, (intType,intType):intType, TgtStr={"-"})
TgtOpr (MulOp, iMul, (intType,intType):intType, TgtStr={"*"})
```

```
TgtOpr (DivOp, iDiv, (intType,intType):intType, TgtStr={"/"})

TgtOpr (AddOp, rAdd, (realType,realType):realType, TgtStr={"+"})
TgtOpr (SubOp, rSub, (realType,realType):realType, TgtStr={"-"})
TgtOpr (MulOp, rMul, (realType,realType):realType, TgtStr={"*"})
TgtOpr (DivOp, rDiv, (realType,realType):realType, TgtStr={"/"})

TgtOpr (PlusOp, iPlus, (intType):intType, TgtStr={"+"})
TgtOpr (NegOp,  iNeg,  (intType):intType, TgtStr={"-"})

TgtOpr (PlusOp, rPlus, (realType):realType, TgtStr={"+"})
TgtOpr (NegOp,  rNeg,  (realType):realType, TgtStr={"-"})

TgtOpr (AddOp, bAdd, (boolType,boolType):boolType, TgtStr={"||"})
TgtOpr (MulOp, bMul, (boolType,boolType):boolType, TgtStr={"&&"})

TgtOpr (NotOp, bNot, (boolType):boolType, TgtStr={"!"})
```

This macro is invoked in definition 29.

The last component associates a value to a property named `TgtStr` of the target operator. The name of the property, its type, and its value may be chosen arbitrarily. Here they are strings that could be used for translating the operators to C. Since, translation is not specified in this text, that property is not further used. However, we have to specify our choice:

OprType.pdl[32]==

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

This macro is attached to a product file.

The target operator signatures as given above would require operands to have exactly those types. E. g. `a + 1` would be illegal if `a` were of type `realType`. The following specification introduces coercion from `intType` to `realType` be applicable if necessary to identify a target operator.

OprType.oil[33]==

```
COERCION cFloat (intType): realType;
```

This macro is attached to a product file.

Type analysis for binary and unary expressions needs to compute the type attribute of the whole expression (the result type of the operation) and the required operand type (the corresponding type of the signature of the identified target operator). The latter may differ from the operand type if coercion is applied. We can obtain these computations from the module roles `BinTgtOpr` and `UnTgtOpr`. We only need to pass the type information to and from the operator symbol as shown below. The module role `ChkOpr` issues a message if no target operator with a suitable signature could be identified. (The key of the target operator could also be obtained by `BinOpr.TgtKey` or `UnOpr.TgtKey`, if necessary e. g. for translation.)

Opr.lido[34]==

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

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

This macro is attached to a product file.

9 Record Types

We now introduce record types to our language in order to demonstrate how composed program defined types are specified. A record type is described by a sequence of field declarations which have the same form as `ObjDecls` used in variable declarations. The notation of variables is extended by an alternative for component selection from a variable.

The following concrete productions are added:

ScopeProp.con[35]==

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

This macro is attached to a product file.

Here is an example program that defines and uses a record variable `rv`: **RecordExamp**[36]==

```
begin
  var   record int i, bool b, real r end rv;
  var   int j, bool c, real s;
  j = rv.i;
  c = rv.b;
  s = rv.r;
end
```

This macro is attached to a product file.

An abstraction of a record type is the sequence of component definitions. The selection construct needs to identify a definition within such a sequence. Hence it is suitable to use a scope for that abstraction. The module `AlgScopeProp` supports using scopes as properties, in particular for describing types.

ScopeProp.specs[37]==

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

This macro is attached to a product file.

A `RecordType` introduces a new type in a program, different from any other type. That concept is represented by the role `TypeDenotation` of the `Typing` module.

In a different language we could have specified that the notation of one record type is equal or compatible to another notation of a record type at a different place in the program, e. g. if they have the same sequence of component (structural equality). In that case we had to describe the rules for equality or compatibility in the `EqualTypes` or `CompatibleTypes` functions; but we would use the same specification as given below. We have to decide this question of type equality for every form of `TypeDenoter` in the language.

The role `RangeScopeProp` of the `AlgScopeProp` module specifies the `RecordType` to be a range. Its scope of component definitions is associated to the `ScopeKey`. The `ScopeKey` is specified to be the type key created by the role `TypeDenotation`.

ScopeProp.lido[38]==

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

SYMBOL RecordType INHERITS
        TypeDenotation, RangeScopeProp
COMPUTE
  SYNT.ScopeKey = SYNT.Type;
  SYNT.GotType = ResetTypeName (SYNT.Type, "record...");
END;
```

This macro is attached to a product file.

The last computation above set the `TypeName` property of the created type for the facility of printing type information we have introduced above. The attribute `GotType` is provided by the `TypeDenotation` role, indicating that all relevant properties of the created type are set. The module then takes care that those properties are set when they are accessed for objects having that type. (The scope property need not be included in that mechanism, because the `AlgScopeProp` module takes care for setting the scope property before using it.

Type analysis for a record component selection is specified in the following way: The `Variable` a selector is applied to has to have a record type. In our case that is decided by checking that the type of the `Variable` has the `Scope` property being set. The selection identifier is to be bound in that scope. We set the attribute `SelectIdent.Scope` to that scope in order to use a role of the `AlgScopeProp` module for binding.

Finally, the type of the whole selection is the type of the bound component:

Selection.lido[39]==

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

  Variable[1].Type = SelectIdent.Type;

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

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

This macro is attached to a product file.

The last symbol computation combines several roles to solve the record selection analysis task: A `SelectIdent` is an identifier occurrence (`IdentOcc`), which is bound in an explicitly determined scope (`IdUseScopeProp`). It is checked whether such a binding is found (`ChkIdUse`). With respect to type analysis, a `SelectIdent` is an occurrence of a typed object identifier (`TypedUseId`, `ChkTypedUseId`, `PrtType`).

10 Type Definitions

We now introduce type definitions to our language. A name can be defined for any `TypeDenoter`. That name can be used to denote the type it stands for.

The following concrete productions are added:

TypeDef.con[40]==

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

This macro is attached to a product file.

Here is an example program with type definitions. It makes use of the facility that identifiers may be defined after their uses: **TypedefExamp**[41]==

```
begin
  var   tt rv;
  type  t tt;
  type  record int i, bool b, real r end t;
  var   int j, bool c, real s;
  var   t rt;
  j = rv.i;
  c = rv.b;
  s = rv.r;
  rt = rv;
end
```

This macro is attached to a product file.

A type definition introduces a new name for a type given by the `TypeDenoter`. In our language we specify that a type definition does not introduce a new type, rather it introduces another name for a type. Hence, there may be many different names for the same type. This view is supported by the roles of the `Typing` module: They just establish a `Defer` relation from the key of the `TypeDefIdent` to the type of the `TypeDenoter`.

Hence, in the following specification we only have to characterize a defining occurrence of a type identifier by the corresponding roles of the name analysis module and those for a defining occurrence of a type identifier (`TypeDefDefId`, `ChkTypeDefDefId`). In the `Declaration` context the `Type` attribute is just passed from the `TypeDenoter` to the `TypeDefIdent`.

TypeDef.lido[42]==

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

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

This macro is attached to a product file.

If we wanted to deviate from the above rules for name equality in case of certain types, we would use the same specification, and would augment the function that checks for equality of type by suitable comparisons, e. g. compare the structure of two record types.

11 Array Types

We now add arrays to our language in order to show how types are represented by freely chosen properties (in contrast to the scope property of record types). We only have to extend the grammar by notations for array type denoters and by indexed variables:

The following concrete productions are added:

Array.con[43]==

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

This macro is attached to a product file.

Here is an example program that uses arrays, records, and type definitions in combination:
ArrayExamp[44]==

```
begin
  var   int k;
  var   int[5] pi, int[5] pj;
  var   record int i, bool b, real[3] r end [2] rv;
  type  bool[4] bt;
  var   bt vbt, bt wbt;
  var   real[6][7] m;
  pi[1] = k;
  vbt = wbt;
  rv[2].b = true;
  rv[1].r[k] = 3.2;
  m[1][k] = 1.0;
end
```

This macro is attached to a product file.

Here, an array type is represented by the type and the number of its elements:

Array.pdl[45]==

```
ElemType:       DefTableKey [KReset];
ElemNo:         int [KReset];
```

This macro is attached to a product file.

An `ArrayType` introduces a new type different from any other (role `TypeDenotation`). Its `Type` attribute is just passed upward.

With respect to equality of type the same considerations apply as were made above for record types.

The properties that characterize the array type are set in a combined computation establishing the postcondition `ArrayType.GotType`, which is essential to ensure correct access of these properties.

ArrayType.lido[46]==

```
SYMBOL ArrayType INHERITS TypeDenotation END;

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

TERM IntNumber: int;

RULE: ArrayType ::= TypeDenoter '[' IntNumber ']' COMPUTE
  ArrayType.GotType =
     ResetElemType
       (KResetElemNo
          (KResetTypeName (ArrayType.Type, "array..."),
           IntNumber),
        TypeDenoter.Type);
END;
```

This macro is attached to a product file.

The type of an indexed variable is the element type of the array variable. The function `TransDefer` has to be applied for the accessed element type, in order to get the type key rather than a key of a type identifier referring to it.

Array.lido[47]==

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

  Expression.ReqType = intType;

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

This macro is attached to a product file.

12 Pointer Types

In this section we introduce pointer types to our language. A type denoted `t !` shall be the type of objects that points to objects of type `t`.

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

The following concrete productions are added:

Pointer.con[48]==

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

This macro is attached to a product file.

Here is an example program that uses these pointer constructs in different contexts: **PointerExamp**[49]==

```
begin
  var   int k;
  var   int! pi, int! pj;
  var   record int i, bool b, real! r end! rv;
  type  record int x, t! next end t;
  var   t l;
  pi = k&;
  pi! = 1;
  pi = pj;
  pi! = pj!;
  rv!.b = true;
  rv!.r! = 3.2;
  l.next!.x = 1;
end
```

This macro is attached to a product file.

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

Pointer.pdl[50]==

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

This macro is attached to a product file.

Types `t !` and `s !` shall be equivalent if `s` and `t` are equivalent, due to type definitions that simply rename a type. (This rule is similar to that of C, but different from that of Pascal where any occurrence of a pointer type denoter introduces a new type.)

As discussed in the section where we introduced record types, we specify a `TypeDenoter` such that a new type representation is created. The `EqualTypes` function is augmented below by rules that implement the desired equivalence of two types that stem from different occurrences of `TypeDenoter`s.

PointerType.lido[51]==

```
SYMBOL PointerType INHERITS TypeDenotation END;

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

RULE: PointerType ::= TypeDenoter '!' COMPUTE
  PointerType.GotType =
    ResetTypeName
      (KResetPointsTo (PointerType.Type, TypeDenoter.Type),
       "pointer...");

  IF (RecursivePtrType (PointerType.Type, PointerType.Type),
  message (ERROR, "recursive pointer type", 0, COORDREF))
  <- INCLUDING Program.GotType;
END;
```

This macro is attached to a product file.

The last computation checks that the pointer type denoted here does not refer to itself via pointer relations only. Note: Although a message is given in such erroneous cases, functions that operate on types must be coded such that they can deal with such types.

We now state how pointer types are checked for equivalence. Our language requires that two pointer type notations shall be considered to be equal types if the types they point to are equal. Such cases where different type keys may represent equal types are not yet captured by the initial frame of the `EqualTypes` function. Instead, it has an insertion point `EqualTypes.h` where code fragments can be inserted, each fragment checking an equivalence rule for a certain kind of types.

Here, the following `.phi` fragment is applicable if the type `t1` is a pointer type. Equivalence holds only if `t2` is a pointer type, too, and the types they point to are equal. Furthermore, we have to make sure that the recursion terminates even in erroneous cases where one of the types is defined to point to itself.

Pointer.EqualTypes.phi[52]==

```
{ /* pointer types: */
  DefTableKey tr1, tr2;
  tr1 = TransDefer (GetPointsTo (t1, NoKey));
  if (tr1 != NoKey)
  { /* tr1 is a pointer type: */
        tr2 = TransDefer (GetPointsTo (t2, NoKey));
        if (tr2 == NoKey)
                return 0; /* tr2 is not a pointer type */
        if (RecursivePtrType (tr1, tr1) ||
            RecursivePtrType (tr2, tr2))
                return 0; /* recursive pointer type */
        return EqualTypes (tr1, tr2);
  }
}/* end pointer types */
```

This macro is attached to a product file.

The `.phi` technique is also used to insert the definition of the function `RecursivePtrType` and its interface declaration:

Pointer.TypeFct.phi[53]==

```
#ifdef PROTO_OK
int RecursivePtrType (DefTableKey orig, DefTableKey chk)
#else
```

```
int RecursivePtrType (orig, chk) DefTableKey orig, chk;
#endif
/* 1 is returned if the type chk directly or indirectly
   points to orig; 0 is returned otherwise.
*/
{ chk = TransDefer (GetPointsTo (chk, NoKey));
  if (chk == NoKey) return 0;
  if (chk == orig)  return 1;
  return RecursivePtrType (orig, chk);
}
```

This macro is attached to a product file.

Pointer.TypeFctHdr.phi[54]==

```
extern int RecursivePtrType ELI_ARG((DefTableKey orig, DefTableKey chk));
```

This macro is attached to a product file.

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

PointerVar.lido[55]==

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

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

This macro is attached to a product file.

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

AddressVar.lido[56]==

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

This macro is attached to a product file.

13 Functions

We now introduce function declarations to our language. A function is characterized by its signature. Function types are an example of types that have properties which are lists of types, the types of the parameters.

We extend the concrete grammar by productions for function declarations:

Function.con[57]==

```
Declaration:    FunctionDecl.
FunctionDecl:   'fun' DefIdent Function ';'.
Function:       FunctionHead Block.
FunctionHead:   '(' Parameters ')' TypeDenoter.
Parameters:     [Parameter // ','].
Parameter:      TypeDenoter DefIdent.
```

This macro is attached to a product file.

Here is an example program that defines some functions. The grammar for function calls and return statements is given below. **FunctionExamp**[58]==

```
begin
  var   int i, int j,
        bool b, bool c,
        real r, real s;

  fun f (int x, real y) real
  begin r = x * y; return r;end;

  fun g (real z) void
  begin r = z; return; end;

  s = f (i+1, 3.4);
  g (f (j, s));
  return;
end
```

This macro is attached to a product file.

A function signature consists of the result type and a list of parameter types. Hence, we use an instance of the `LidoList` module in order to compose such type lists. (Note: If we had more than one mode of parameter passing, the abstraction of a parameter in the function signature would be a pair: parameter passing mode and parameter type.)

Function.specs[59]==

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

This macro is attached to a product file.

The use of the module requires a particular name for the non-existing list element:

Function.head[60]==

```
#define NoDefTableKey NoKey
```

This macro is attached to a product file.

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

Function.pdl[61]==

```
ParamTypes:     DefTableKeyList; "DefTableKeyList.h"
ResultType:     DefTableKey [KReset];
```

This macro is attached to a product file.

We first consider the name analysis aspect of a function declaration. The `Function` subtree is a range where the parameter definitions are valid. The function `Block` is nested in that range. Since the `DefIdent` for parameters are already completely specified for name analysis, we need only:

FunctionScope.lido[62]==

```
SYMBOL Function INHERITS RangeScope END;
```

This macro is attached to a product file.

Now we consider a function declaration as a definition of a typed object, and apply the same specification pattern as used for variable declarations. Furthermore, each `Parameter` is also a `TypedDefinition`. There is no problem in nesting definitions of a typed objects this way.

FunctionDecl.lido[63]==

```
SYMBOL FunctionDecl INHERITS TypedDefinition END;

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

RULE: Function ::= FunctionHead Block COMPUTE
  Function.Type = FunctionHead.Type;
END;

SYMBOL Parameter INHERITS TypedDefinition END;

RULE: Parameter ::= TypeDenoter DefIdent COMPUTE
  Parameter.Type = TypeDenoter.Type;
  DefIdent.IsVariable = 1;
END;
```

This macro is attached to a product file.

Now we specify how the type of a function is composed. The `FunctionHead`, which contains the signature, is treated as a `TypeDenotation` for a function type. Its `ResultType` property is obtained from the `TypeDenoter` of the result type. The `ParamTypes` property is the list of parameter types. That list is simply composed from the `Root` and `Elem` role of the `LidoList` module instantiated for lists of `DefTableKeys`.

FunctDeclType.lido[64]==

```
SYMBOL FunctionHead INHERITS TypeDenotation END;
```

```
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter COMPUTE
  FunctionHead.GotType =
    ResetParamTypes
      (KResetResultType
         (KResetTypeName (FunctionHead.Type, "function..."),
          TypeDenoter.Type),
       Parameters.DefTableKeyList);
END;

SYMBOL Parameters INHERITS DefTableKeyListRoot END;
SYMBOL Parameter INHERITS DefTableKeyListElem END;

RULE: Parameter ::= TypeDenoter DefIdent COMPUTE
  Parameter.DefTableKeyElem = TypeDenoter.Type;
END;
```

This macro is attached to a product file.

Function calls are integrated in the expression syntax of our language. We chose a very general form of an `Expression` to denote the function to be called. That allows us to later expand the language by expressions which yield a function. That feature does not create additional problems for type analysis.

We also introduce return statements into our language.

Call.con[65]==

```
Expression:     Expression '(' Arguments ')'.
Arguments:      [Argument // ','].
Argument:       Expression.

Statement:      'return' ';'.
Statement:      'return' Expression ';'.
```

This macro is attached to a product file.

Type analysis for a function call is straight-forward: We access the `ResultType` from the type of the function `Expression`; it is the type of the call. The list of parameter types is passed to the `Arguments`. There it is decomposed into its elements using the list decomposition roles of the `LidoList` module, `DeListRoot` and `DeListElem`. That mechanism also works if in erroneous cases the lengths of the parameter list and the structural argument list are different.

Call.lido[66]==

```
SYMBOL Arguments INHERITS DefTableKeyDeListRoot END;
RULE: Expression ::= Expression '(' Arguments ')' COMPUTE
  Expression[1].Type =
        TransDefer (GetResultType (Expression[2].Type, NoKey));

  Arguments.DefTableKeyList =
        GetParamTypes (Expression[2].Type, NULLDefTableKeyList);
END;

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

This macro is attached to a product file.

The computation of the ReqType attribute above completely captures type checking for each argument. For the function Expression we only have to check whether it really yields a function, the first check below. The other two checks issue messages if the lengths of the lists are different.

CallCheck.lido[67]==

```
RULE: Expression ::= Expression '(' Arguments ')' COMPUTE
  Expression[2].ReqType = Expression[2].Type;

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

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

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

This macro is attached to a product file.

A return statement refers to the immediately enclosing function declaration. We have to check that a value of a type compatible to the result type is returned if the latter is not `void`. A return from the outermost program level is considered as if the program was a `void` function.

Return.lido[68]==

```
SYMBOL Program COMPUTE
  SYNT.Type = KResetResultType (NewKey(), voidType);
END;

ATTR resType: DefTableKey;

RULE: Statement ::= 'return' ';' COMPUTE
  .resType =
     TransDefer
       (GetResultType
          (INCLUDING (FunctionDecl.Type, Program.Type),
           NoKey))
     <- INCLUDING Program.GotType;

  IF (NOT (EqualTypes (.resType, voidType)),
  message (ERROR, "return value required", 0, COORDREF));
END;

RULE: Statement ::= 'return' Expression ';' COMPUTE
  .resType =
     TransDefer
       (GetResultType
          (INCLUDING (FunctionDecl.Type, Program.Type),
           NoKey))
     <- INCLUDING Program.GotType;

  IF (EqualTypes (.resType, voidType),
  message (ERROR, "return value not allowed", 0, COORDREF));

  Expression.ReqType = .resType;
END;
```

This macro is attached to a product file.

14 Function Types

We finally extend our language towards the orthogonal use of functions, i. e. wherever a typed object is allowed it can have a function type. For that purpose we add another `TypeDenoter` which denotes function types.

The following concrete productions are added:

FunctType.con[69]==

```
TypeDenoter:    FunctionType.
FunctionType:   '(' ParamTypes '->' TypeDenoter ')'.
ParamTypes:     [ParamType // ','].
ParamType:      TypeDenoter.
```

This macro is attached to a product file.

Here is an example program that defines a function type and a higher order function: **FctType-Examp**[70]==

```
begin
  fun mul (int x, real y) real
  begin return x * y; end;

  fun add (int x, real y) real
  begin return x + y; end;

  type (int, real -> real) fct;

  fun apply2 (real z, fct ff) real
  begin return ff (2, z); end;

  var real r;

  r = apply2 (3.1, add);
  r = apply2 (3.1, mul);
end
```

This macro is attached to a product file.

The introduction of function types to our language requires that we state rules for the structural equivalence of function types. Otherwise the type rules would not allow one, for example, to assign a function to a function variable, or to pass it as an argument to a function. The involved types are represented by different type keys.

Hence, we extend the insertion point of the `EqualTypes` function, as in the case of pointer types. The inserted code fragment is applicable if the type `t1` is a function type, i. e. the `ResultType` property is set. If type `t2` is a function type, too, both types are decomposed and its components are checked for being pairwise equal.

Function.EqualTypes.phi[71]==

```
{ /* function types: */
  DefTableKey tr1, tr2;
  tr1 = TransDefer (GetResultType (t1, NoKey));
  if (tr1 != NoKey)
  { /* tr1 is a function type */
    DefTableKeyList pr1, pr2;
    /* ensure termination: */
    if (InFunctionType (t1, t1)) return 0;
    if (!EqualTypes (tr1, GetResultType (t2, NoKey)))
       return 0; /* tr2 is not a function type */
    if (InFunctionType (t2, t2)) return 0;
    pr1 = GetParamTypes (t1, NULLDefTableKeyList);
    pr2 = GetParamTypes (t2, NULLDefTableKeyList);
    if (0 == CompDefTableKeyList (pr1, pr2, CmpTypes))
       return 1;
  }
}/* end function types */
```

This macro is attached to a product file.

Here again, we have to make sure that the check terminates if, in erroneous cases, the function type refers to itself using a function `InFunctionType`. Furthermore, the comparison of the parameter type lists using the function `CompDefTableKeyList` requires a specific function for element comparison, here `CmpTypes`. Both function definitions and the header of `InFunctionType` are supplied to the two insertion points of the type checking module using the `.phi` mechanism.

Function.TypeFct.phi[72]==

```
#ifdef PROTO_OK
```

```
int InFunctionType (DefTableKey t1, DefTableKey t2)
#else
int InFunctionType (t1, t2) DefTableKey t1, t2;
#endif
/* On exit
 *    returns 1 if t2 is a function type that contains t1;
 *    otherwise 0 is returned.
 */
{ DefTableKey t; DefTableKeyList tl;
  t1 = TransDefer (t1);
  t2 = TransDefer (t2);
  t = TransDefer (GetResultType (t2, NoKey));
  if (t != NoKey) {
        if (t == t1) return 1;
        if (InFunctionType (t1, t)) return 1;

        tl = GetParamTypes (t2, NULLDefTableKeyList);
        while (tl) {
                t = TransDefer (HeadDefTableKeyList (tl));

                if (t == t1) return 1;
                if (InFunctionType (t1, t)) return 1;

                tl = TailDefTableKeyList (tl);
        }
  }
  return 0;
}

static
#ifdef PROTO_OK
int CmpTypes (DefTableKey t1, DefTableKey t2)
#else
int CmpTypes (t1, t2) DefTableKey t1, t2;
#endif
{
  return (EqualTypes (t1, t2) ? 0 : 1);
}
```

This macro is attached to a product file.

Function.TypeFctHdr.phi[73]==

```
extern int InFunctionType ELI_ARG((DefTableKey t1, DefTableKey t2));
```

This macro is attached to a product file.

The composition of the function type representation follows the same pattern as used in function declarations.

We need a check that prohibits a function type directly or indirectly refer to itself, using the function `InFunctionType`.

FunctType.lido[74]==

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

SYMBOL FunctionType INHERITS TypeDenotation END;

RULE: FunctionType ::= '(' ParamTypes '->' TypeDenoter ')' COMPUTE
  FunctionType.GotType =
    ResetParamTypes
      (KResetResultType
         (KResetTypeName (FunctionType.Type, "function..."),
          TypeDenoter.Type),
       ParamTypes.DefTableKeyList);

  IF (InFunctionType (FunctionType.Type, FunctionType.Type),
  message (ERROR, "recursive function type", 0, COORDREF))
  <- INCLUDING Program.GotType;
END;

SYMBOL ParamTypes INHERITS DefTableKeyListRoot END;
SYMBOL ParamType INHERITS DefTableKeyListElem END;

RULE: ParamType ::= TypeDenoter COMPUTE
  ParamType.DefTableKeyElem = TypeDenoter.Type;
END;
```

This macro is attached to a product file.