Tutorial on Type Analysis

Compiler and Programming Language Group
University of Paderborn
Faculty for Electrical Engineering, Computer Science and Mathematics
Department of Computer Science
33098 Paderborn, Germany

Table of Contents

Overview

This tutorial is an introduction to the solution of the type analysis task required for programming language implementation. It demonstrates many aspects of that task which may occur in programming languages or in special purpose languages. The tutorial demonstrates how the components of Eli's type analysis library are used to solve that task. The tutorial proceeds from basic aspects, such as declarations, typed entities, 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 type analysis tasks, 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 entities, and error reports where examples violate specified language properties.

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*.

Chapter 1 specifies a small language kernel and solves the name analysis and the type analysis task for it. The focus of the following chapters 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 type 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 and 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,
        int r, int s;
  i = 1;
  b = true;
  r = 3;
  j = i;
  c = b;
  s = r;
end
```
      This macro is attached to a product file.

Structure and notation of the kernel language is specified here by its abstract syntax.

Abstract Kernel syntax[2]==

```
RULE: Program       ::=    Block END;
RULE: Block         ::=    'begin' Declarations Statements 'end' END;
RULE: Declarations LISTOF Declaration END;
RULE: Statements   LISTOF Statement END;

RULE: Declaration  ::=    'var' ObjDecls ';' END;
RULE: ObjDecls     LISTOF ObjDecl  END;
RULE: ObjDecl      ::=    TypeDenoter DefIdent END;
RULE: TypeDenoter  ::=    'int' END;
RULE: TypeDenoter  ::=    'bool' END;
RULE: TypeDenoter  ::=    'void' END;

RULE: Statement    ::=    Variable '=' Expression ';' END;
RULE: Statement    ::=    Expression ';' END;

RULE: Expression   ::=    Variable END;
RULE: Expression   ::=    IntNumber END;
RULE: Expression   ::=    'true' END;
RULE: Expression   ::=    'false' END;

RULE: Variable     ::=    UseIdent END;

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

This macro is invoked in definition 16.

Concrete syntax rules corresponding to the `LISTOF` constructs above, specifications of the notations of identifiers, literals, and comments are given in the appendix.

1.1 Basic Scope Rules

The basic task of name analysis is consistent renaming. For each identifier occurrence a `Key` attribute is computed such that it identifies a program entity uniquely. `Keys` are used to associate properties to program entities and to retrieve those properties in different contexts. The symbols `DefIdent`, `UseIdent` distinguish defining and used identifier occurrences.

The scope rules of a language determine how identifier occurrences are bound to program entities. 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:

Basic scope module[3]==

    $/Name/AlgScope.gnrc:inst
    This macro is invoked in definition 14.

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 by any identifier occurrence.

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

Kernel scope rules[4]==

    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;
    This macro is invoked in definition 16.

Erroneous programs may violate the scope rules in two different situations:

- A particular applied identifier occurrence has no valid defining identifier occurrence.
- There are more than one defining identifier occurrences for the same identifier in a 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:

Message support[5]==

```
$/Tech/Strings.specs
$/Prop/OccCnt.gnrc:inst
```
This macro is invoked in definition 14.

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

The `OccCnt` module provides computations that count how often an entity 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 entity occurs more than once in the `ChkUnique` context it is multiply defined.

Scope checks[6]==

```
SYMBOL UseIdent 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 invoked in definition 16.

1.2 Types in the Kernel Language

We use the modules `Typing` to support type analysis. As we are going to specify structural equivalence for some kinds of type, we also instantiate the module `StructEquiv`. *Type analysis module*[7]==

```
$/Type/Typing.gnrc:inst
$/Type/StructEquiv.fw
```
This macro is invoked in definition 14.

So, we have to adopt the modules' strategy for representing types:

Types are represented by `DefTableKey`s. Such a key is created for each program construct which denotes a particular type. The unknown type is represented by `NoKey`.

The kernel language has only language defined types: `int`, `bool`, and `void`. Each of them is represented by a known key. Here we introduce only the key for the type `void`, as the other types occur in operator specification, and are introduced there: *Language defined type keys*[8]==

```
voidType -> IsType = {1};
```
This macro is invoked in definition 15.

All type keys have a property `IsType`, which distinguishes them from keys representing entities other than types. Usually the property `IsType` is not set or accessed by user specifications. Module roles ensure that they are properly used.

The following computations set the `Type` attributes of the constructs that denote languge defined types: *Language defined types*[9]==

```
RULE: TypeDenoter ::= 'int'  COMPUTE TypeDenoter.Type = intType;  END;
RULE: TypeDenoter ::= 'bool' COMPUTE TypeDenoter.Type = boolType; END;
RULE: TypeDenoter ::= 'void' COMPUTE TypeDenoter.Type = voidType; END;
```
This macro is invoked in definition 16.

Further forms of `TypeDenoter`s for user defined types are specified in subsequent sections.

We now consider a variable declaration as an example for a language construct that defines a typed entity. In our language a variable declaration may define several variables. An `ObjDecl` states the type and the name for each of them.

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

Declarations[10]==

```
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 invoked in definition 16.

The module roles `TypedUseId` classifies a used name of a typed entity, and causes the attribute `TypedUseId.Type` to be set to the type defined for that entity. The corresponding check role issues messages if that classification is violated:

Typed identifiers[11]==

```
SYMBOL UseIdent INHERITS TypedUseId, ChkTypedUseId END;
```
This macro is invoked in definition 16.

In order to report some results of the type analysis we associate two properties to every type key: a string value `TypeName` and the number of the line where the type is introduced. (The latter will become more significant when user defined types are defined for the language.)

Output properties[12]==

```
TypeName: CharPtr; "Strings.h"
TypeLine: int;

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

intType ->  TypeLine = {0};
boolType -> TypeLine = {0};
voidType -> TypeLine = {0};
```

This macro is invoked in definition 15.

For every used identifier the name and the defining line of its type is printed: *Kernel output*[13]==

```
SYMBOL UseIdent INHERITS PrtType END;

SYMBOL PrtType COMPUTE
  printf ("line %d Type %s defined in line %d\n", LINE,
          GetTypeName (THIS.Type, "no type name"),
          GetTypeLine (THIS.Type, 0))
  <- INCLUDING  Program.TypeIsSet;
END;
```

This macro is invoked in definition 16.

Kernel.specs[14]==

Basic scope module[3]
Message support[5]
Type analysis module[7]

This macro is attached to a product file.

Kernel.pdl[15]==

Language defined type keys[8]
Output properties[12]

This macro is attached to a product file.

Kernel.lido[16]==

Abstract Kernel syntax[2]
Kernel scope rules[4]
Scope checks[6]
Language defined types[9]
Declarations[10]
Typed identifiers[11]
Kernel output[13]

This macro is attached to a product file.

Kernel.gla[17]==

Token notation[137]

This macro is attached to a product file.

Kernel.con[18]==

Concrete Kernel syntax[135]

This macro is attached to a product file.

Oprand.sym[19]==

Expression mapping[136]

This macro is attached to a product file.

2 Type Checking in Expressions

Expressions consist of typed names and literals and of operators that are applied to operands of certain types and yield a result of a certain type. Determining the types of expressions and checking the related type rules of the language is a significant subtask of type analysis. The type rules of languages are usually formulated in terms of concepts like "type of program constructs and entities", "signature of operators", "operator overloading", "type conversion". They have common and well-understood meaning for type analysis in general. Of course, the type rules established for a particular language instantiate these concepts in a specific way, e.g. define a specific set of operators with their signature and state which conversions may be applied to resolve overloading.

Eli's type analysis module `Expression` provides reusable roles and computations to formulate the language specific instantiation of the concepts mentioned above. The type analysis for expressions is generated from such a specification.

Expression module[20]==

```
$/Type/Expression.gnrc:inst
```
This macro is invoked in definition 26.

This module carries out type analysis on expression trees, which are subtrees made up of connected expression nodes. An expression node is a node representing a program construct that yields a value of a certain type. The module provides the role `ExpressionSymbol` to be inherited by symbols that are expression symbols in that sense:

Expression symbols[21]==

```
SYMBOL Expression INHERITS ExpressionSymbol END;
SYMBOL Variable INHERITS ExpressionSymbol END;
```
This macro is invoked in definition 27.

The type of each expression node is characterized by two attributes: `ExpressionSymbol.Type` describes the type of the values this expression may yield. `ExpressionSymbol.Required` may be used to specify that the upper context requires the expression to yield a value of a particular type. As `ExpressionSymbol.Required` is used to compute `ExpressionSymbol.Type`, it may not depend on the `Type` attribute.

Expression symbols may occur in different contexts with respect to the structure of the expression trees: root contexts, leaf contexts, and inner contexts. The module provides different computational roles for those contexts. In leaf contexts the type of the leaf expression must be stated using the computational role `PrimaryContext`. Note that in the third role below the expression node is a leaf with respect to the expression tree, although the context has one subtree, that is not an expression node:

Leaf nodes[22]==

```
RULE: Expression ::= IntNumber COMPUTE
  PrimaryContext (Expression, intType);
END;

RULE: Expression ::= 'true' COMPUTE
  PrimaryContext (Expression, boolType);
END;
```

```
RULE: Expression ::= 'false' COMPUTE
  PrimaryContext (Expression, boolType);
END;


RULE: Variable ::= UseIdent COMPUTE
  PrimaryContext (Variable, UseIdent.Type);
END;
```
This macro is invoked in definition 27.

The computational role `TransferContext` is used for contexts that have an expression node on the left-hand side and one on the right-hand side, and both have the same type properties:

Transfer nodes[23]==
```
RULE: Expression ::= Variable COMPUTE
  TransferContext (Expression, Variable);
END;
```
This macro is invoked in definition 27.

The node representing an assignment statement has two children. Both are considered as roots of expression trees. For the `Variable` the assignment context does not impose any restriction on its type; hence, nothing is specified for that node:

Assignment[24]==
```
RULE: Statement ::= Variable '=' Expression ';' COMPUTE
  RootContext (Variable.Type, , Expression);
  Indication (assignOpr);
END;
```
This macro is invoked in definition 27.

explain the purpose of assignOpr??

The `Expression` on the right-hand side of the assignment is required to yield a value of the type of the `Variable`.

An expression in the role of a statement is another example for a root context. On execution the value of the expression will just be discarded. Hence, there is no requirement on its type to be stated or checked: *Expression statement*[25]==
```
RULE: Statement ::=  Expression ';' END;
```
This macro is invoked in definition 27.

Expression.specs[26]==

Expression module[20]
This macro is attached to a product file.

Expression.lido[27]==

Expression symbols[21]
Leaf nodes[22]
Transfer nodes[23]
Assignment[24]
Expression statement[25]
This macro is attached to a product file.

3 Operator Overloading

We here extend our language by binary and unary operators in order to demonstrate type analysis for expressions with overloaded operators.

Operators are overloaded in our language, i.e. an operator symbol like `+` may denote one of several operations, e.g. integer addition or logical disjunktion (or). The distinction is made using the types of the operands. Hence, we associate to an operator symbol like `+` an indication like `AddOp`, which represents a set of operators, like `iAdd, bOr`.

Each of the following rules associates an indication name to the attribute `BinOpr.Indic`. (The indication names are introduced below.)

Operator Indications[28]==

```
RULE: BinOpr ::= '+' COMPUTE BinOpr.Indic = AddOp; END;
RULE: BinOpr ::= '-' COMPUTE BinOpr.Indic = SubOp; END;
RULE: BinOpr ::= '*' COMPUTE BinOpr.Indic = MulOp; END;
RULE: BinOpr ::= '/' COMPUTE BinOpr.Indic = DivOp; END;

RULE: UnOpr ::= '+' COMPUTE UnOpr.Indic = PlusOp; END;
RULE: UnOpr ::= '-' COMPUTE UnOpr.Indic = NegOp; END;
RULE: UnOpr ::= '!' COMPUTE UnOpr.Indic = NotOp; END;
```
     This macro is invoked in definition 34.

For each of the operator indications at least one meaning is specified by one of the following operation descriptions. The first component of an operation description relates it to an indication representing the operator symbol, the second component is a unique name for the operation.

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

All names are automatically introduced as names for definition table keys. They may be used explicitly in specifications to distinguish operations, or to associate properties to them.

For each language defined operator its signature is specified; operators that have the same signature can be comprised in one definition: *Oil Operation Signatures*[29]==

```
OPER
  iAdd, iSub, iMul, iDiv (intType,intType):intType;
  iPlus, iNeg            (intType):intType;
  bOr, bAnd              (boolType,boolType):boolType;
  bNot                   (boolType):boolType;
```
     This macro is invoked in definition 35.

Next, we associate a set of operators to every indication. Here, for example the `AddOp` is overloaded with three operations: `iAdd` and `bOr`, and `MulOp` is overloaded with `iMul` and `bAnd`. All other indications have singleton sets: *Oil indications*[30]==

```
INDICATION
  AddOp:  iAdd, bOr;
  SubOp:  iSub;
  MulOp:  iMul, bAnd;
  DivOp:  iDiv;
```

```
        PlusOp: iPlus;
        NegOp:  iNeg;
        NotOp:  bNot;
```
This macro is invoked in definition 35.

The operation signatures as given above require operands to have exactly those types. E.g.
`a + 1` is illegal if `a` was of type `boolType`.

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 types of operands (the
corresponding type of the signature of the identified target operator). The latter may differ
from the type of the operand in case that coercion is applied. We obtain these computations
from the `Expression` module.

Operator contexts[31]==

```
      SYMBOL BinOpr INHERITS OperatorSymbol END;

      RULE: Expression ::= Expression BinOpr Expression COMPUTE
        DyadicContext (Expression[1], BinOpr, Expression[2], Expression[3]);
      END;

      SYMBOL UnOpr INHERITS OperatorSymbol END;
      RULE: Expression ::= UnOpr Expression COMPUTE
        MonadicContext (Expression[1], UnOpr, Expression[2]);
      END;
```
This macro is invoked in definition 36.

The key of the identified operation could be obtained by `BinOpr.Oper` or `UnOpr.Oper`, if
necessary e.g. for translation.

Operator.con[32]==

Expression syntax[138]

This macro is attached to a product file.

Operator.sym[33]==

Operators[139]

This macro is attached to a product file.

Indications.lido[34]==

Operator Indications[28]

This macro is attached to a product file.

Operator.oil[35]==

Oil Operation Signatures[29]
Oil indications[30]

This macro is attached to a product file.

Operator.lido[36]==

Operator contexts[31]

This macro is attached to a product file.

4 Type Conversion

This chapter introduces type conversion to our language. We say, a value of a certain type `t` is converted into a corresponding value of some other type `s`. For example, a conversion of integral values into floating point values is defined for many languages. We consider such a conversion be executed by a conversion operator that has a signature `t->s`. We call a conversion `coercion` if the application of a conversion operator is determined implicitly, for example in the process of overloading resolution.

In order to demonstrate type conversion, we extend our language by a second arithmetic type for floating point values an call the type `real`.

The type representation is extended by:

Real type representation[37]==

```
    realType -> TypeName = {"real"};
```
This macro is invoked in definition 44.

We add a new type denoter to the language

Real type denoter[38]==

```
    RULE: TypeDenoter ::= 'real' COMPUTE TypeDenoter.Type = realType; END;
```
This macro is invoked in definition 45.

and introduce literals of type `real`:

Real literals[39]==

```
    RULE: Expression ::= RealNumber COMPUTE
      PrimaryContext (Expression, realType);
    END;
```
This macro is invoked in definition 45.

Now we extend the set of operator specifications by operators for the type `real`:

Real operators[40]==

```
    OPER
      rAdd  (realType,realType):realType;
      rSub  (realType,realType):realType;
      rMul  (realType,realType):realType;
      rDiv  (realType,realType):realType;

      rPlus (realType):realType;
      rNeg  (realType):realType;
```
This macro is invoked in definition 46.

We specify that the `real` operators overload the corresponding ones for the type `int` by adding them to the corresponding indication:

Real operators overload[41]==

```
    INDICATION
      AddOp: rAdd;
      SubOp: rSub;
      MulOp: rMul;
      DivOp: rDiv;
```

```
      PlusOp: rPlus;
      NegOp:  rNeg;
```
This macro is invoked in definition 46.

Now we want to allow that overloading resolution takes conversion from `int` to `real` into account. That means in an expression like `a + 1` the operand types need not match exactly to the signature of a `+` operator, if coercion could convert the operand types into those required by the signature. In particular `a` could have type `real`. In that case coercion from `int` to `real` would be applied to `1` in order to use the `real` addition operator.

So, we define such a coercion operator `iTor` with the signature `int->real`:

Predefined Coercion Operator[42]==

```
      COERCION
        iTor (intType):realType;
```
This macro is invoked in definition 46.

Finally we reconsider the type rules for assignments. We want to allow to have an `int` variable on the left-hand side and a `real` expression on the right, say `i = 3.4;` That means the result of the expression is to be converted to an `int` value, which is then assigned to the variable.

For that purpose we specify a conversion operator `rToi` with the signature `real->int`, and associate it to the operator indication `assignOpr` which has been introduced for the assignment context:

Assignment Conversion Operator[43]==

```
      OPER
         rToi (realType):intType;
      INDICATION
        assignOpr: rToi;
```
This macro is invoked in definition 46.

Note: The conversion operator `rToi` is only applicable in a context that is chacterized by the indication `assignOpr`, it is NOT applied as a coercion when resolving overloaded operators.

RealType.pdl[44]==

Real type representation[37]

This macro is attached to a product file.

RealType.lido[45]==

Real type denoter[38]
Real literals[39]

This macro is attached to a product file.

OperatorExtensions.oil[46]==

Real operators[40]
Real operators overload[41]
Assignment Conversion Operator[43]
Predefined Coercion Operator[42]

This macro is attached to a product file.

5 Record Types

We introduce record types to our language in order to demonstrate how composed user
defined types are specified. A record type is described by a sequence of field declarations
which have the same semantics as `ObjDecls` used in variable declarations. A notation for
variables is added that allows to select a component from a variable.

Here is an example program that defines and uses a record variable named `rv`:

RecordExamp[47]==

```
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.

The following productions describe record types and component selections:

Abstract record syntax[48]==

```
RULE: TypeDenoter ::= RecordType END;
RULE: RecordType  ::='record' ObjDecls 'end' END;


RULE: Variable    ::= Variable '.' SelectIdent END;
RULE: SelectIdent ::= Ident END;
```
This macro is invoked in definition 62.

An abstraction of a record type is the sequence of component definitions, each consisting
of a type and a name. A `RecordType` describes such a type abstraction. It inherits the
module role `TypeDenotation`:

Type denoter[49]==

```
SYMBOL RecordType INHERITS TypeDenotation END;

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

RULE: RecordType ::= 'record' ObjDecls 'end' COMPUTE
  .GotTypeProp =
    ORDER (
      ResetTypeName (RecordType.Type, "record..."),
      ResetTypeLine (RecordType.Type, LINE));

END;
```
This macro is invoked in definition 62.

The last computation above sets the properties `TypeName` and `TypeLine` of the created
type for the facility of printing type information we have introduced above. The attribute

`GotTypeProp` represents that state. It is used in another instance of this RULE context below, where further properties are associated to the type.

The construct for component selection, e.g. `rv.i`, demonstrate a typical situation where type ananlysis and name analysis depend on each other: The type of the variable `rv` has a property, which is a scope; it is used to lookup a binding for the selector `i`. Hence we instantiate the name analysis module `ScopeProp`, which supports scopes as properties. It is adapted to the needs of type analysis by the module `TypeDep`: *Scope property module*[50]==

```
$/Name/ScopeProp.gnrc:inst
$/Type/TypeDep.gnrc:inst
```

This macro is invoked in definition 60.

The role `ExportRange` of the `ScopeProp` module specifies the `RecordType` to be a range that may export its bindings to be lookedup outside of that range, e.g. in component selections. Its scope of component definitions is associated to the `ScopeKey`. The `ScopeKey` is specified to be the type key created by the role `TypeDenotation`: *Range*[51]==

```
SYMBOL RecordType INHERITS ExportRange COMPUTE
  SYNT.ScopeKey = SYNT.Type;
END;
```

This macro is invoked in definition 62.

5.1 Type Equivalence

As record types have non-trivial abstractions, the question arises under which circumstances two record types are the same. Consider the following examples: **RecordEqual**[52]==

```
begin
  var   record int i, bool b, real r end va;
  var   record int i, bool b, real r end vc;
  var   record int j, bool k, real l end vd;
  va = vc;
  va = vd;
end
```

This macro is attached to a product file.

Typing rules of the language have to state which of the variables `va`, `vc`, and `vd` have the same type, and which of the assignments are correct. Languages usually apply one of two different typing rules:

The first rule states that every occurrence of a description of a record type (or of any other compound type) introduces a type different from all other types, even from those that are equally notated. Under this rule all three variables have different types. This rule is called name equivalence, because every type description gets a name - explicitly or implicitly, as in this example -and types are distingushed by their names.

The second rule states that two types are equal if their abstractions are equal; i.e. the sequences of components coincide elementwise in the types and names of components. In the above example `va` and `vc` have the same types. This rule is called structural equivalence.

In case of structural equivalence the type rules of the language may define precisely, which type properties belong to the abstraction that is used to determine type equivalence. For

example, the rule could state that the types of the record components belong to the abstraction, and the names of the components do not belong to it. In that case all four variables of the above example would have the same type.

The type analysis library provides a module `StructEquiv` that extends the `Typing` module, such that any of these these variants of equivalence rules can be supported: *Struct equiv module*[53]==

```
$/Type/StructEquiv.fw
```
This macro is invoked in definition 60.

In this language stuctural equivalence is specified, such that for record types only the sequence of types, but not the names of components are relevant for structural type equivalence.

The following computation in the RULE context of a record type denotation specifies which properties of a record type are considered for the check whether two types are equivalent. Here we state two rules:

First, a record type can only be equivalent to a type that is a record type, too. For that purpose we introduce a key `RecordClass` that identifies the category of record types: *Type class*[54]==

```
RecordClass;
```
This macro is invoked in definition 61.

The rule computation `AddTypeToBlock` below associates every record type to that initial set `RecordClass`. The equivalence check will then partition it as far as necessary into subsets of record types which are equivalent.

Second, two record types `s` and `t` are equivalent if the types of their fields are pairwise equivalent in the given order. For that purpose a list of component types is computed `ObjDecls.OpndTypeList` using roles of the `LidoList` module and given as the third argument of `AddTypeToBlock`.

Beyond type equivalence, our language requires further checks on type structures. So, the list of component types is also associated as a property `ComponentTypes` to the type key by a function `VResetComponentTypes` that yields the property value as its result: *Component type property*[55]==

```
ComponentTypes: DefTableKeyList [VReset]; "DefTableKeyList.h"
```
This macro is invoked in definition 61.

PropLib module[56]==

```
$/Prop/PropLib.fw
```
This macro is invoked in definition 60.

The attribute `RecordType.GotType` states that all properties of the record type are associated to its key. Hence, a dependence on the attribute `GotTypeProp` computed above is added here. *Type equality computation*[57]==

```
RULE: RecordType ::= 'record' ObjDecls 'end' COMPUTE
  RecordType.GotType =
      AddTypeToBlock
         (RecordType.Type, RecordClass,
          VResetComponentTypes (RecordType.Type, ObjDecls.OpndTypeList))
      <- .GotTypeProp;
```

```
END;

SYMBOL ObjDecls INHERITS OpndTypeListRoot END;
SYMBOL ObjDecl INHERITS OpndTypeListElem END;

SYMBOL ObjDecl COMPUTE
  SYNT.DefTableKeyElem = SYNT.Type;
END;
```
This macro is invoked in definition 62.

5.2 Qualified Names

A record component selection of the form `Variable.SelectIdent` is considered as a qualified name: The `SelectIdent` is an applied occurrence of an identifier that is qualified by the `Variable` preceeding the dot. Its type is expected to have a scope property that has a binding for that identifier.

`Variable.SelectIdent` is a leaf of an expression tree. Its type is determined by the type of `SelectIdent`, as specified using the `PrimaryContext` computation. *Selection expression*[58]==

```
RULE: Variable ::= Variable '.' SelectIdent COMPUTE
  PrimaryContext (Variable[1], SelectIdent.Type);
END;
```
This macro is invoked in definition 62.

`SelectIdent` combines roles of name analysis and type analysis: It is a qualified identifier use (`QualIdUse`). The role requires that the attribute `SelectIdent.ScopeKey` is computed. A module computation accesses the (`Scope` property from it, stores it in `SelectIdent.Scope` and searches a binding for the identifier; the role `ChkQualIdUse` gives a message if the scope exists, but no binding is found. A user computation is required to check whether the type has a scope property.

The roles `TypedUseId`, `ChkTypedUseId`, and `PrtType` determine, check, and output the type of the `SelectIdent`.

Selection types[59]==

```
SYMBOL SelectIdent INHERITS
        QualIdUse, ChkQualIdUse, IdentOcc,
        TypedUseId, ChkTypedUseId, PrtType
END;

RULE: Variable ::= Variable '.' SelectIdent COMPUTE
  SelectIdent.ScopeKey = Variable[2].Type;

  IF (EQ (SelectIdent.Scope, NoEnv),
  message (ERROR, "selection applied to non record type",
           0, COORDREF));
END;
```
This macro is invoked in definition 62.

Record.specs[60]==

Scope property module[50]
Struct equiv module[53]
PropLib module[56]

This macro is attached to a product file.

Record.pdl[61]==

Type class[54]
Component type property[55]

This macro is attached to a product file.

Record.lido[62]==

Abstract record syntax[48]
Type denoter[49]
Range[51]
Type equality computation[57]
Selection expression[58]
Selection types[59]

This macro is attached to a product file.

6 Array Types

We now add array types to our language. We specify that two array types are structural equivalent if their element types are equivalent, and if the types have the same number of elements. Hence, type equivalence is not only determined by the component types.

Here is an example program that uses arrays, records, and type definitions in combination:

ArrayExamp[63]==

```
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.

We extend the grammar by notations for array type denoters and by indexed variables:

Abstract array syntax[64]==

```
RULE: TypeDenoter ::= ArrayType END;
RULE: ArrayType   ::= TypeDenoter '[' ArraySize ']' END;
RULE: ArraySize   ::= IntNumber END;

RULE: Variable    ::= Variable '[' Expression ']' END;
```

This macro is invoked in definition 74.

In this language an array type is described by two properties: the element type and the number of elements: *Array type properties*[65]==

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

This macro is invoked in definition 73.

In the context of a type denotation for an **ArrayType** the two properties of the type are set together with the **TypeName** to indicate the array type in the output. The attribute **GotTypeProp** specifies that these properties are set.

Array type denoter[66]==

```
SYMBOL ArrayType INHERITS TypeDenotation END;

RULE: ArrayType ::= TypeDenoter '[' ArraySize ']' COMPUTE
  .GotTypeProp =
     ORDER
       (ResetElemType (ArrayType.Type, TypeDenoter.Type),
```

```
            ResetElemNo   (ArrayType.Type, ArraySize.Size),
            ResetTypeName (ArrayType.Type, "array..."),
            ResetTypeLine (ArrayType.Type, LINE));
    END;

    TERM IntNumber: int;

    SYMBOL ArraySize: Size: int;

    RULE: ArraySize ::= IntNumber COMPUTE
      ArraySize.Size = IntNumber;
    END;

    RULE: TypeDenoter ::= ArrayType COMPUTE
      TypeDenoter.Type = ArrayType.Type;
    END;
```
This macro is invoked in definition 74.

Finally it is stated that array elements of type void are not allowed. We can not simply
compare voidType and the type key, because TypeDenoter.Type not necessarily contains
the final element type; it may be related to it. The final type key is obtained by the function
FinalType in a state that is characterized by INCLUDING Program.TypeIsSet. *Array check
element type* [67]==

```
    RULE: ArrayType ::= TypeDenoter '[' ArraySize ']' COMPUTE
      IF (EQ (FinalType (TypeDenoter.Type), voidType),
          message (ERROR, "Wrong element type", 0, COORDREF))
      <- INCLUDING Program.TypeIsSet;
    END;
```
This macro is invoked in definition 74.

Two array types are equivalent if and only if their element types are equivalent and if they
have the same number of elements.

In order to state the equivalence with respect to array sizes, we establish a bijective mapping
between any array size that occurs in the program and a definition table key. That number
mapping is computed by turning an array size into an identifier and then binding that
identifier in a scope that serves just this purpose.

Size mapping[68]==

```
    $/Tech/MakeName.gnrc:inst
    $/Name/CScope.gnrc+instance=SizeMap :inst
```
This macro is invoked in definition 72.

Array size mapping[69]==

```
    SYMBOL ArraySize INHERITS SizeMapIdDefScope END;

    RULE: ArraySize ::= IntNumber COMPUTE
      ArraySize.Sym = IdnNumb (0, IntNumber);
    END;
```

This macro is invoked in definition 74.

The `ArraySize.Key` serves as the initial set of potential equivalent array types; it is used as the second argument of the RULE computation `AddTypeToBlock`. The type of the element may contribute to type equivalence of array types. Hence, the third argument of `AddTypeToBlock` is a singleton list, which is also set as the `ComponentTypes` property of the array type:

Array type equivalence[70]==

```
RULE: ArrayType ::= TypeDenoter '[' ArraySize ']' COMPUTE
  ArrayType.GotType =
    AddTypeToBlock
     (ArrayType.Type, ArraySize.Key,
      VResetComponentTypes
        (ArrayType.Type, SingleDefTableKeyList (TypeDenoter.Type)))
     <- .GotTypeProp;
END;
```

This macro is invoked in definition 74.

Type analysis in the context of an indexed variable is specified as a join of three expression subtrees: `Variable[1]`, the left-hand side of the rule is a leaf of an expression tree. `PrimaryContext` is used to state that its type is the `ElemType` property of `Variable[2]`.

`Variable[2]`, which yields the array, is considered to be the root of an expression subtree. No requirements are specified. It has to be checked explicitly that its type is an array type.

The subcript expression is a separate expression subtree. It has to be of type int, as specified by its `Required` attribute.

Indexing[71]==

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

  IF (EQ (GetElemType (Variable[2].Type, NoKey), NoKey),
    message (ERROR, "Not an array", 0, COORDREF));

  Expression.Required = intType;
END;
```

This macro is invoked in definition 74.

Array.specs[72]==

Size mapping[68]

This macro is attached to a product file.

Array.pdl[73]==

Array type properties[65]

This macro is attached to a product file.

Array.lido[74]==

⟨ *Abstract array syntax*[64]
Array type denoter[66]
Array check element type [67]
Array size mapping[69]
Array type equivalence[70]
Indexing[71]
This macro is attached to a product file.

7 Union Types

We introduce union types to our language in order to demonstrate how subtype relations and their coercions are specified. A union type is described by a sequence of type denotations, which constitute the subtypes of the specified union type. A value of one of the subtypes can be coerced to the union type. A value of a union type can be treated as a value of one of the subtypes using a cast operation or a case statement.

Here is an example program that defines and uses a union variable named `rv`:

UnionExamp[75]==

```
begin
  var   union int, bool end rv;
  var   int j, bool c;
  rv = 42; rv = true;
  j = <int> rv;
  case rv of
    int t: j = t;
    bool t: c = t;
  end
end
```
This macro is attached to a product file.

In the `case` statement the `case` expression has a union type. Each case declares a variable of a subtype of that union type. The branch which corresponds to the current type of the `case` expression is selected, its variable is initialized with the value of the `case` expression, and the statement is executed.

The following productions describe union types, type casts, and `case` statements: *Abstract union syntax*[76]==

```
RULE: TypeDenoter ::= UnionType END;
RULE: UnionType  ::= 'union' UnitedTypes 'end' END;
RULE: UnitedTypes LISTOF UnitedType END;
RULE: UnitedType ::= TypeDenoter END;

RULE: Expression ::= '<' TypeDenoter '>' Expression END;

RULE: Statement ::= CaseStmt END;
RULE: CaseStmt ::= 'case' Expression 'of' Cases 'end' END;
RULE: Cases LISTOF Case END;
RULE: Case ::= ObjDecl ':' Statement END;
```
This macro is invoked in definition 88.

The following computations introduce a type denoter for union types and associate properties for test output to it: In order to check whether a type is a union type, as required for example in a case statement, we introduce a property `IsUnionType`. *Is union type*[77]==

```
IsUnionType: int;
```
This macro is invoked in definition 87.

Union type denoter[78]==

```
SYMBOL UnionType INHERITS TypeDenotation END;

RULE: UnionType ::= 'union' UnitedTypes 'end' COMPUTE
  .GotTypeProp =
      ORDER (
        ResetIsUnionType (UnionType.Type, 1),
        ResetTypeName (UnionType.Type, "union..."),
        ResetTypeLine (UnionType.Type, LINE));
END;


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

This macro is invoked in definition 88.

For the comparison of union types stuctural equivalence is specified, such that the fact that it is a union type and the sequence of subtypes are relevant for type equality. `UnionClass` is the the set containing all union types for initialization of the equivalence check.

Union type class[79]==

```
UnionClass;
```

This macro is invoked in definition 87.

The `UnionClass` and the sequence `UnitedTypes.OpndTypeList` are used as arguments of `AddTypeToBlock` to specify type equivalence of union types. Property `ComponentTypes` is set accordingly:

Union type equality computation[80]==

```
RULE: UnionType ::= 'union' UnitedTypes 'end' COMPUTE
  UnionType.GotType =
      AddTypeToBlock
        (UnionType.Type, UnionClass,
         VResetComponentTypes (UnionType.Type, UnitedTypes.OpndTypeList))█
      <- .GotTypeProp;
END;

SYMBOL UnitedTypes INHERITS OpndTypeListRoot END;
SYMBOL UnitedType  INHERITS OpndTypeListElem END;

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

This macro is invoked in definition 88.

Note, that here the order of the subtypes in the type denoter is relevant for type equality. If that is not desired, one could for example sort the list of the component types in a canonical order before using it as an argument of `AddTypeToBlock`.

For each union type we introduce two groups of conversion operators: A widening coercion from each subtype type to the union type, and a down cast from the union type to each subtype. For the latter an indication has to be introduced: *Downcast indication*[81]==

```
DownCast;
UnionWiden;
```
This macro is invoked in definition 87.

As a pair of operators has to be introduced for each subtype, the context of the subtype denoter is the right place to do it. The coercion operator is not created explicitly; it is only stated that the subtype is `Coercible` to the union type. The down cast conversion is introduced as a `MonadicOperator`:

Widening coercion computation[82]==

```
SYMBOL UnitedType INHERITS OperatorDefs COMPUTE
  SYNT.GotOper =
    ORDER
      (Coercible (UnionWiden, THIS.Type, INCLUDING UnionType.Type),
       MonadicOperator
         (DownCast, NewKey(),
          INCLUDING UnionType.Type, THIS.Type));
END;
```
This macro is invoked in definition 88.

The context of the down cast construct imposes a requirement on the type of the operand expression. Any conversion operator of the `DownCast` indication can be applied to the operand expression, in addition to any coercions, to satisfy this requirement:

Down cast[83]==

```
RULE: Expression ::= '<' TypeDenoter '>' Expression COMPUTE
  CastContext (Expression[1], , Expression[2], TypeDenoter.Type);
  Indication (DownCast);
END;
```
This macro is invoked in definition 88.

In a `case` statement it is required that the `case` expression has a union type:

Union case statement[84]==

```
SYMBOL CaseStmt: Type: DefTableKey;

RULE: CaseStmt ::= 'case' Expression 'of' Cases 'end' COMPUTE
  CaseStmt.Type = Expression.Type;

  IF (NOT (GetIsUnionType (Expression.Type, 0)),
  message (ERROR, "Case expression must have a union type",
           0,COORDREF))
  <- INCLUDING Program.TypeIsSet;
END;
```
This macro is invoked in definition 88.

Each branch of a `case` statement forms a range for the declaration of the variable that gets the value of the `case` expression if that case is selected. It is required that the type of the

variable is a subtype of the type of the case expression. We here require that it is coercible to the type of the **case** expression, although that is not quite exact. *Union case*[85]==

```
SYMBOL Case INHERITS RangeScope END;

RULE: Case ::= ObjDecl ':' Statement COMPUTE
  IF (NOT (IsCoercible
             (ObjDecl.Type, INCLUDING CaseStmt.Type)),
  message (ERROR, "Must be a subtype of the case expression",
             0, COORDREF))
  <- INCLUDING Program.GotType;
END;
```
This macro is invoked in definition 88.

In other contexts **ObjDecl** occurs in a CHAIN. To avoid an error message on missing a chain start we apply the role **OpndTypeListRoot** here, which has the CHAINSTART, although that role is not needed:

Union CHAIN workaraound[86]==

```
SYMBOL Cases INHERITS OpndTypeListRoot END;
```
This macro is invoked in definition 88.

Union.pdl[87]==

> *Is union type*[77]
> *Union type class*[79]
> *Downcast indication*[81]

This macro is attached to a product file.

Union.lido[88]==

> *Abstract union syntax*[76]
> *Union type denoter*[78]
> *Union type equality computation*[80]
> *Widening coercion computation*[82]
> *Down cast*[83]
> *Union case*[85]
> *Union case statement*[84]
> *Union CHAIN workaraound*[86]

This macro is attached to a product file.

Union.con[89]==

> *Concrete union syntax*[144]

This macro is attached to a product file.

8 Functions

This chapter introduces definitions and calls of parameterized functions. Type analysis has to check that the signature of a function call matches the signature of the called function, and that functions return a value of the specified type.

Here is an example program that defines some functions. The grammar for function calls and return statements is given below.

FunctionExamp[90]==

```
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.

We first extend the grammar by productions for function declarations: *Abstract function syntax*[91]==

```
RULE: Declaration ::=  FunctionDecl END;
RULE: FunctionDecl ::= 'fun' DefIdent Function ';' END;
RULE: Function ::=    FunctionHead Block END;
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter END;
RULE: Parameters LISTOF Parameter END;
RULE: Parameter ::= TypeDenoter DefIdent END;
```
This macro is invoked in definition 100.

A function type is characterized by its signature, i.e. the sequence of the types of its parameters and the result type. (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.)

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`s of parameters are already completely specified for name analysis, we need only:

Function range[92]==

```
SYMBOL Function INHERITS RangeScope END;
```
This macro is invoked in definition 100.

Now we consider a function declaration as a definition of a typed entity, 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 typed entities this way.

Function declaration types[93]==

```
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;
END;
```
This macro is invoked in definition 100.

Next, 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.

Function type[94]==

```
SYMBOL FunctionHead INHERITS TypeDenotation, OperatorDefs END;
```
This macro is invoked in definition 100.

Furthermore, a function declaration introduces an operator. This is indicated by the role `OperatorDefs`. The computation `ListOperator` creates a new operator, identified by `FunctionHead.Type`. The types of the parameters together with the result type `TypeDenoter.Type` form its signature.

Function signature[95]==

```
RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter COMPUTE
  FunctionHead.GotOper +=
      ListOperator (
        FunctionHead.Type,
        FunctionHead.Type,
        Parameters.OpndTypeList,
        TypeDenoter.Type);
END;

SYMBOL Parameters INHERITS OpndTypeListRoot END;
SYMBOL Parameter INHERITS OpndTypeListElem END;

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

This macro is invoked in definition 100.

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:

Abstract call syntax[96]==

```
RULE: Expression ::=    Expression '(' Arguments ')' END;
RULE: Arguments  LISTOF Argument END;
RULE: Argument   ::=    Expression END;


RULE: Statement  ::=    'return' ';' END;
RULE: Statement  ::=    'return' Expression ';' END;
```

This macro is invoked in definition 100.

Type analysis for a function call is straight-forward: A call is treated as an operation which takes the arguments as operands. `Expression[2]` yields the function to be called. Its type provides the operator indication, which may be overloaded with several operations, as stated in the context of the function definition. The precoined computation `ListContext` connects the expression subtree of the arguments with `Expression[1]` representing the result.

Call types[97]==

```
SYMBOL Arguments INHERITS OpndExprListRoot END;
SYMBOL Argument INHERITS OpndExprListElem END;

RULE: Expression ::= Expression '(' Arguments ')' COMPUTE
  ListContext (Expression[1], , Arguments);
  Indication (Expression[2].Type);

  IF(BadOperator,
    message
      (ERROR,
       "Call does not match the functions' signatures",
       0, COORDREF));
END;
```

This macro is invoked in definition 100.

The following context connects the `Argument` node with the expression subtree forming the actual parameter. If they had the same type properties, we would have used a `TransferContext` computation. However, in our language we want to allow that the type of the `Expression` need not match exactly the type required for the `Argument` as specified in the signature of the function. As in assignments it shall be allowed that the expression yields a value of type `real` which then is converted to an `int` value if required by the function signature, e.g. in `f(3.4)`.

Hence, we use a `ConversionContext` which allowes to connect the `Argument` via an operator with the `Expression` node. The indication `assignOpr` is specified for this context. It states that the same conversion operators as in assignments (i.e. `rToi`) and all coercion operators

(i.e. `iTor`) may be used to convert the result of the `Expression` to the type of the `Argument`, if necessary:

Arguments[98]==

```
    RULE: Argument ::= Expression COMPUTE
      ConversionContext (Argument, , Expression);
      Indication (assignOpr);
    END;
```
This macro is invoked in definition 100.

A return statement refers to the immediately enclosing function declaration. It has to be checked that a value of a type is returned that is compatible to the result type, if the latter is not `void`. A return from the outermost program level is considered as if the program was a `void` function. Conversions that are additionally applicable are specified in the same way as in the `Argument` context above.

The attribute value `Function.ResultType` stems from the context of a type denotation. Hence, its value may not be used directly in a compare with a type key as `voidType`. The function `FinalType` has to access the related type key, and the precondition INCLUDING `Program.TypeIsSet` has to be stated.

Return statements[99]==

```
    ATTR ResultType: DefTableKey;

    RULE: Statement ::= 'return' Expression ';' COMPUTE
      RootContext (
        INCLUDING (Function.ResultType, Program.ResultType), , Expression);
      Indication (assignOpr);
    END;

    RULE: Statement ::= 'return' ';' COMPUTE
      IF (NOT (EQ (voidType,
                   FinalType (
                       INCLUDING (Function.ResultType,
                                  Program.ResultType)))),
      message (ERROR, "return value required", 0, COORDREF))
      <- INCLUDING Program.TypeIsSet;
    END;

    SYMBOL Program COMPUTE
      SYNT.ResultType = voidType;
    END;

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

    RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter COMPUTE
      FunctionHead.ResultType = TypeDenoter.Type;
    END;
```

This macro is invoked in definition 100.

Function.lido[100]==

> *Abstract function syntax*[91]
> *Abstract call syntax*[96]
> *Function declaration types*[93]
> *Function range*[92]
> *Function type*[94]
> *Function signature*[95]
> *Call types*[97]
> *Arguments*[98]
> *Return statements*[99]
>
> This macro is attached to a product file.

Function.con[101]==

> *Function declaration syntax*[140]
> *Call syntax*[141]
>
> This macro is attached to a product file.

9 Type Definitions

This chapter introduces type definitions to the language. A name can be defined for any `TypeDenoter` and can be used to denote that type.

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

TypedefExamp[102]==

```
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.

The following productions are added to the grammar:

Abstract type declaration syntax[103]==

```
RULE: Declaration  ::= 'type' TypeDenoter TypeDefIdent ';' END;
RULE: TypeDefIdent ::= Ident END;

RULE: TypeDenoter  ::= TypeUseIdent END;
RULE: TypeUseIdent ::= Ident END;
```
This macro is invoked in definition 113.

A type definition introduces a new name for a type given by the `TypeDenoter`. We distinguish between defining occurrences `TypeDefIdent` used occurrences `TypeUseIdent` of type names.

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. Furthermore, even two `TypeDenoter` that differ in certain aspects may denote the same type. This view can be supported by the roles of the `StructEquiv` module: For each kind of types it is stated which of its properties distinguish two types of that kind (see record types or array types).

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`.

Type declaration computation[104]==

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

```
END;

RULE: Declaration ::= 'type' TypeDenoter TypeDefIdent ';' COMPUTE
  TypeDefIdent.Type = TypeDenoter.Type;
END;
```
This macro is invoked in definition 113.

Used occurrences of type identifiers are characterized by the module roles `TypeDefUseId`
and `ChkTypeDefUseId`, and by the roles that characterize used indentifier occurrences of
any kind:

Used type identifiers[105]==
```
SYMBOL TypeUseIdent INHERITS
    IdUseEnv, IdentOcc, ChkIdUse,
    TypeDefUseId, ChkTypeDefUseId END;

RULE: TypeDenoter ::= TypeUseIdent COMPUTE
  TypeDenoter.Type = TypeUseIdent.Type;
END;
```
This macro is invoked in definition 113.

A language that has facilities to define names for types and allows identifier uses before
their definitions, opens the possibility to define types recursively, e.g.

```
type record int i, rt x end rt;
```
In many languages such a type `rt` would be disallowed, because a value of type `rt` may
not contain another value of the same type. However, if the type of the component `x` was
defined to be `pointer to rt` instead of `rt`, then a useful list type would be defined.

This example illustrates that the existence of type definitions may cause the need to specify
which recursive type definitions are considered legal for a certain language. In our language,
as defined so far, any recursion in a type definition is considered to be disallowed. However,
the situation changes when there are types, like pointer types, such that that recursion is
allowed when it passes through such a type.

Hence, we introduce three properties: `IsRecursiveType`, `IsNotRecursiveType` indicates
whether a type is illegally recursive. `AllowRecurType` indicates that recursion through such
a type is allowed. The latter will be set when such types are introduced, i.e. in the chapter
on pointer types: *Check recursive types properties*[106]==
```
IsRecursiveType: int;
IsNotRecursiveType: int;
AllowRecurType: int;
```
This macro is invoked in definition 112.

Such a check is performed by a function `ChkRecursiveType` which is applied to a type `t`
and recursively walks through the component types of `t`. If it reaches `t` again (without
having passed through a type that allows recursion), the type is indicated to be illegally
recursive.

Check for recursive types[107]==
```
SYMBOL TypeDefIdent COMPUTE
  IF (ChkRecursiveType (THIS.Type),
```

```
        message (ERROR, CatStrInd ("Recursively defined type: ",
                                    THIS.Sym),
                 0, COORDREF))
      <- INCLUDING Program.GotAllTypes;
    END;
```
This macro is invoked in definition 113.

The implementation of the type walk algorithm uses a property that indicates whether a type is currently visited by the algorithm: *Visiting property*[108]==

```
    Visiting: int;
```
This macro is invoked in definition 112.

The following C module implements an algorithm that walks recursively through the components of a type to check whether the type is defined illegally recursively.

RecTypeChk.head[109]==

```
    #include "RecTypeChk.h"
```
This macro is attached to a product file.

RecTypeChk.h[110]==

```
    #include "deftbl.h"
    extern int ChkRecursiveType (DefTableKey tp);
```
This macro is attached to a product file.

RecTypeChk.c[111]==

```
    #include "pdl_gen.h"
    #include "StructEquiv.h"
    #include "Typing.h"

    #ifdef TEST
    #define TEST 1
    #include <stdio.h>
    #endif

    static DefTableKey origType;


    int VisitCompOfType (DefTableKey node, DefTableKey component)
    /* This is the function used by the type walk that checks
       for recursive types. It is called for every visit from a type node
       to one of its components. 5 cases are distinguished, as explained below:
    */
    {
    #ifdef TEST
      printf ("visit at %s (%d) component %s (%d)\n",
           GetTypeName (node, "no name"),
           GetTypeLine (node, 0),
           GetTypeName (component, "no name"),
           GetTypeLine (component, 0));
    #endif
```

```
   if (GetAllowRecurType (component, 0))
      /* do not visit a component type that allows recursion */
      return 1;
   else {
      if (FinalType (component) == FinalType (origType)) {
        /* the type under investigation contains itself on a path
           that does not lead through a type which allows for
           recursion
        */
        ResetIsRecursiveType (origType, 1);
        ResetIsRecursiveType (component, 1);
        return 2; /* terminate walk */
      }
      if (GetIsRecursiveType (component, 0)) {
      /* The component type of the node is already
         recognized to be recursive
      */
        ResetIsRecursiveType (origType, 1);
        return 2; /* terminate walk */
      }
      if (GetVisiting (component, 0)) {
        /* This component lies on a non-pointer cycle
           not involving the original type under investigation
        */
        ResetIsRecursiveType (origType, 1);
        ResetIsRecursiveType (component, 1);
        return 2; /* terminate walk */
      }
   }
   return 0; /* visit this component */
}

int RecWalkType (DefTableKey currType)
/*
    Every direct or indirect component type of tp is visited, unless
    the call VisitCompOfType (curr, comp) shortcuts the walk.
    If it returns
      0: comp is visited
      1: comp is skipped
      2: the walk is terminated
*/
{ DefTableKeyList compseq;
  DefTableKey compType;
  int visitRes;
  currType = FinalType (currType);

  /* Do not visit a node, that is currently visited: */
```

```
        if (GetVisiting (currType, 0)) return 1;
        ResetVisiting (currType, 1);

        /* consider all component types: */
        for (compseq = GetComponentTypes (currType, NULLDefTableKeyList);
             compseq != NULLDefTableKeyList;
             compseq = TailDefTableKeyList (compseq)) {
          compType = FinalType (HeadDefTableKeyList (compseq));

          /* Skip non-type: */
          if (compType == NoKey) continue;

          /* Visit this component: */
          visitRes = VisitCompOfType (currType, compType);

          /* The component visit indicates how to continue: */
          switch (visitRes) {
            case 0:              /* dive into the component */
               if (RecWalkType (compType)) {
                  /* the type walk is to be terminated */
                  visitRes = 2;
                  goto ret;
               };
               break;
            case 1:              /* do not dive into the component */
               break;
            case 2:              /* terminate the walk */
               visitRes = 2;
               goto ret;
            default:;
            }
            /* iteration of components continues */
        }
        visitRes = 0; /* components elaborated */
   ret:
     ResetVisiting (currType, 0);
     return visitRes;
   }


   int ChkRecursiveType (DefTableKey tp)
   /* on entry:
        The results of the type equivalence analysis must be computed and
        stored in the type data base.
        tp represents a type.
      method:
        A walk through the type structure of tp is initiated,
```

```
          and then executed.
          VisitCompOfType (curr, comp) is called whenever
          a direct component comp of the type curr is visited.
          origType stores the type for which the recursion check is initiated.▮
          Every type that is found to be illegally recursive is marked by
          the property IsRecursiveType. It is also used to shortcut
          the walk through the type structure.
        on exit:
          1 is returned if the type tp directly or indirectly
          has tp as a component type, and the path to it is not
          legal for recursion
          0 is returned otherwise.
      */
      {
        tp = FinalType (tp);
        origType = tp;
        if (GetIsRecursiveType (tp, 0)) return 1;
        if (GetIsNotRecursiveType (tp, 0)) return 0;

        /* the result is not yet known: */
        RecWalkType (tp);

        return GetIsRecursiveType (tp, 0);
      }
```

This macro is attached to a product file.

TypeDef.pdl[112]==

   Check recursive types properties[106]
   Visiting property[108]

This macro is attached to a product file.

TypeDef.lido[113]==

   Abstract type declaration syntax[103]
   Type declaration computation[104]
   Used type identifiers[105]
   Check for recursive types[107]

This macro is attached to a product file.

10 Pointer Types

In this chapter we introduce pointer types to our language. The notation `t !` denotes a type for values that point to values of type `t`.

A new `Variable` notation is introduced: In `v !` the dereferencing operator `!` is applied to the variable `v`, which must have a pointer type. The result of the operation is the value that `v` points to.

A pointer value of type `t !` is created by execution of a generator `new t`, where `t` is a type denotation.

Here is an example program that uses these pointer constructs in different contexts:

PointerExamp[114]==

```
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 = new int;
  pi! = 1;
  pi = pj;
  pi! = pj!;
  rv!.b = true;
  rv!.r! = 3.2;
  l.next!.x = 1;
  l.next = nil;
end
```
   This macro is attached to a product file.

The following productions are added to the grammar:

Abstract pointer syntax[115]==

```
RULE: TypeDenoter ::= PointerType END;
RULE: PointerType ::= TypeDenoter '!' END;

RULE: Variable    ::= Variable '!' END;
RULE: Expression  ::= 'nil' END;
RULE: Expression  ::= Generator END;
RULE: Generator   ::= 'new' TypeDenoter END;
```
   This macro is invoked in definition 125.

There are two constructs which introduce a pointer type. The first one is a denoter for a pointer type. Two monadic operators are created for each pointer type: One is applied to a pointer and yields the value pointed to, the other yields the reference of an entity instead of its value. The dereferencing operators of all pointer types are overloaded on the indication `DerefOpr`, correspondingly all operators that prevent dereferencing are overloaded on the indication `RefOpr`. We also introduce an artificial type for the `nil` symbol: *Pointer operators*[116]==

```
DerefOpr;
RefOpr;
NilOpr;
nilType -> IsType = {1};
```
This macro is invoked in definition 124.

Creating these pairs of operators for a pointer type establishes the condition
`PointerType.GotOper`, which is a precondition for operator identification. Furthermore,
we state that the type of the `nil` symbol is coercible to each pointer type.

Pointer type denotation[117]==

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


SYMBOL PointerType INHERITS TypeDenotation, OperatorDefs END;


RULE: PointerType ::= TypeDenoter '!' COMPUTE
  PointerType.GotOper +=
    ORDER
      (Coercible (NilOpr, nilType, PointerType.Type),
       MonadicOperator
         (DerefOpr, NewKey(), PointerType.Type, TypeDenoter.Type),
       MonadicOperator
         (RefOpr, NewKey(), TypeDenoter.Type, PointerType.Type));
END;


RULE: Expression  ::= 'nil' COMPUTE
  PrimaryContext (Expression, nilType);
END;
```
This macro is invoked in definition 125.

A generator also introduces a pointer type. The `TypeDenoter` states which is the type
pointed to. Generators may occur as operand in expressions:

Generator[118]==

```
SYMBOL Generator INHERITS TypeDenotation, OperatorDefs END;

RULE: Generator ::= 'new' TypeDenoter COMPUTE
  Generator.GotOper +=
    ORDER (
    MonadicOperator
       (DerefOpr, NewKey(), Generator.Type, TypeDenoter.Type),
    MonadicOperator
       (RefOpr, NewKey(), TypeDenoter.Type, Generator.Type));
END;

RULE: Expression ::= Generator COMPUTE
  PrimaryContext (Expression, Generator.Type);
END;
```

Types `t !`, `s !` and the types created by `new t` and `new s` are all considered to be equivalent in our language, if the types `s` and `t` are equivalent, with respect to renaming and to equivalence rules for the particular type categories.

We use the facilities of the `StructEquiv` module to specify such structural type equivalence for pointer types. In particular two conditions are specified for types `a` and `b` to be equivalent: Both have to be of the kind `PointerClass`, and their sequences of component types have to be elementwise equivalent, in this case the single type pointed to:

EqualPtrTypes.lido[119]==

```
RULE: PointerType ::= TypeDenoter '!' COMPUTE
  PointerType.GotType =
    AddTypeToBlock
      (PointerType.Type, PointerClass,
       VResetComponentTypes
         (PointerType.Type, SingleDefTableKeyList (TypeDenoter.Type)))
  <- .moreTypeProperies;
END;


RULE: Generator ::= 'new' TypeDenoter COMPUTE
  Generator.GotType =
    AddTypeToBlock
      (Generator.Type, PointerClass,
       VResetComponentTypes
         (Generator.Type, SingleDefTableKeyList (TypeDenoter.Type)))
  <- .moreTypeProperies;
END;
```

The `PointerClass` is a unique key used to distinguish this kind of types from other kinds, e.g. array types:

Pointer type equality[120]==

```
PointerClass;
```

Pointer types are to be treated especially when types are checked for equivalence: On the one hand, a type is allowed to be recursively defined if the recursion goes through a pointer component, for example in `type record int i, rec! p end rec;`. That is why we associate the property `AllowRecurType` to the pointer type, together with the properties defining the output for types.

Pointer types allow recursion[121]==

```
RULE: PointerType ::= TypeDenoter '!' COMPUTE
  .moreTypeProperies =
    ORDER
      (ResetTypeName (PointerType.Type, "pointer..."),
       ResetTypeLine (PointerType.Type, LINE),
       ResetAllowRecurType (PointerType.Type, 1));
END;
```

```
RULE: Generator ::= 'new' TypeDenoter COMPUTE
  .moreTypeProperies =
    ORDER
      (ResetTypeName (Generator.Type, "pointer..."),
       ResetTypeLine (Generator.Type, LINE),
       ResetAllowRecurType (Generator.Type, 1));
END;
```
This macro is invoked in definition 125.

On the other hand, we have to check that pointer types are not defined directly recursively, or indirectly recursively s.t. only pointer types are involved:

```
type p1! p1;
type p2! p3;
type p3! p2;
```
In the example above all three type are pairwise equivalent.

Recursion check for pointer types[122]==

```
RULE: PointerType ::= TypeDenoter '!' COMPUTE
IF (EQ (FinalType (PointerType.Type), FinalType (TypeDenoter.Type)),
  message (ERROR, "Recursively defined pointer type",
             0, COORDREF))
  <- INCLUDING Program.TypeIsSet;
END;
```
This macro is invoked in definition 125.

For the dereferencing operation applied to a `Variable` we specify that in the following context a suitable operator that overloads the `DerefOpr` indication is applicable:

Pointer variable[123]==

```
RULE: Variable ::= Variable '!' COMPUTE
  MonadicContext (Variable[1], , Variable[2]);
  Indication (DerefOpr);

  IF(BadOperator,
    message(ERROR,"Dereferencing not allowed", 0, COORDREF));
END;
```
This macro is invoked in definition 125.

Pointer.pdl[124]==

Pointer type equality[120]
Pointer operators[116]
This macro is attached to a product file.

Pointer.lido[125]==

Abstract pointer syntax[115]
Pointer type denotation[117]
Generator[118]
Pointer types allow recursion[121]
Recursion check for pointer types[122]
Pointer variable[123]

This macro is attached to a product file.

Pointer.con[126]==

    Concrete pointer syntax[143]

    This macro is attached to a product file.

11 Function Types

We finally extend our language towards the orthogonal use of functions, i.e. wherever a typed entity is allowed it can have a function type. In particular, evaluation of an expression may yield a function, which may be called, assigned to a variable, passed as an argument, or returned as a function result. For that purpose it is sufficient to add another `TypeDenoter` which denotes function types. New notations for expressions are not needed.

Here is an example program that defines a function type and a higher order function:

FctTypeExamp[127]==

```
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 following productions are added to the grammer:

Abstract function type syntax[128]==

```
RULE: TypeDenoter  ::=    FunctionType END;
RULE: FunctionType ::=    '(' ParamTypes '->' TypeDenoter ')' END;
RULE: ParamTypes   LISTOF ParamType END;
RULE: ParamType    ::=    TypeDenoter END;
```
This macro is invoked in definition 133.

The specifications for `FunctionType`s exactly correspond to those for `FunctionHead`s in the context of function declarations. An Operator is created, that has a signature as given by the types of the parameters and of the result:

Function type denotation[129]==

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

SYMBOL FunctionType INHERITS TypeDenotation, OperatorDefs END;

RULE: FunctionType ::= '(' ParamTypes '->' TypeDenoter ')' COMPUTE
```

```
      FunctionType.GotOper +=
         ListOperator (
           FunctionType.Type,
           FunctionType.Type,
           ParamTypes.OpndTypeList,
           TypeDenoter.Type);

    .moreTypeProperies =
       ORDER
         (ResetTypeName (FunctionType.Type, "function..."),
          ResetTypeLine (FunctionType.Type, LINE));
   END;
```

This macro is invoked in definition 133.

The introduction of function types to our language allows programs to use values which represent functions. They have a function type which must fit to the type required in the context. For example, the `apply2 (3.1, add)` passes the function `add` as an argument of the called function `apply2`. Hence, the type of the declared function `add` must be equivalent to the type required for the second parameter of `apply2` (or coercible under type rules for parameter, as specified in the chapter on functions).

In this case we have to specify structural equivalence of function types, in oder to let the type rules allow such uses of functions. If we would specify name equivalence instead, then for the above example, the signature of the function declaration and the type `fct` specified for the second parameter of `apply2` are different notations of types. They would be considered not to be name equivalent; but, they are structural equivalent.

Structural type equivalence is specified for denotations of function types that either occur in a type denotation or as the signature of a declared function. We state that two types `a` and `b` are equivalent if both have the kind `FunctionClass`, and the component types, which are the types of the parameters and of the result, are elementwise equivalent:

Function type equivalence[130]==

```
    RULE: FunctionHead ::= '(' Parameters ')' TypeDenoter COMPUTE
      FunctionHead.GotType =
        ORDER (
          AddTypeToBlock (
             FunctionHead.Type, FunctionClass,
             VResetComponentTypes
               (FunctionHead.Type,
                ConsDefTableKeyList
                  (TypeDenoter.Type, Parameters.OpndTypeList))),
             ResetTypeName (FunctionHead.Type, "function..."),
             ResetTypeLine (FunctionHead.Type, LINE));
    END;

    RULE: FunctionType ::= '(' ParamTypes '->' TypeDenoter ')' COMPUTE
      FunctionType.GotType =
        AddTypeToBlock
          (FunctionType.Type, FunctionClass,
```

```
            VResetComponentTypes
              (FunctionType.Type,
               ConsDefTableKeyList
                 (TypeDenoter.Type, ParamTypes.OpndTypeList)))
      <- .moreTypeProperies;
    END;

    SYMBOL ParamTypes INHERITS OpndTypeListRoot END;
    SYMBOL ParamType INHERITS OpndTypeListElem END;

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

This macro is invoked in definition 133.

Function class[131]==

```
    FunctionClass;
```

This macro is invoked in definition 132.

We require for our language, that a function type `f` may not directly or indirectly have a component type `f`, unless the recursion passes through a pointer type. The check is specified in the context of type definitions.

FunctionType.pdl[132]==

Function class[131]

This macro is attached to a product file.

FunctionType.lido[133]==

Abstract function type syntax[128]
Function type denotation[129]
Function type equivalence[130]

This macro is attached to a product file.

FunctionType.con[134]==

Function type syntax[142]

This macro is attached to a product file.

12 Appendix: Syntax

12.1 Concrete Kernel Syntax

Concrete Kernel syntax[135]==

```
Declarations:   Declaration*.
Declaration:    'var' ObjDecls ';'.
ObjDecls:       [ObjDecl // ','].
Statements:     Statement*.

Expression:     Factor.
Factor:         Operand.
Operand:        IntNumber.
Operand:        RealNumber.
Operand:        'true'.
Operand:        'false'.
Operand:        Variable.
```
This macro is invoked in definition 18.

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:

Expression mapping[136]==

```
Expression ::= Factor Operand.
```
This macro is invoked in definition 19.

The notation of identifiers, numbers, and comments is chosen as in Pascal:

Token notation[137]==

```
Ident:          PASCAL_IDENTIFIER
IntNumber:      PASCAL_INTEGER
RealNumber:     PASCAL_REAL
                PASCAL_COMMENT
```
This macro is invoked in definition 17.

12.2 Concrete Expression Syntax

Expression syntax[138]==

```
Expression:     Expression AddOpr Factor.
Factor:         Factor MulOpr Operand.
Operand:        MonOpr Operand.
Operand:        '(' Expression ')'.
AddOpr:         '+' / '-'.
MulOpr:         '*' / '/'.
MonOpr:         '+' / '-' / '!'.
```
This macro is invoked in definition 32.

The following specification unifies the binary operators that have different precedences into one symbol class `BinOpr` of the abstract syntax.

Operators[139]==

```
BinOpr ::= AddOpr MulOpr.
UnOpr  ::= MonOpr.
```
This macro is invoked in definition 33.

12.3 Concrete Function Syntax

Function declaration syntax[140]==

```
Parameters:     [Parameter // ','].
```
This macro is invoked in definition 101.

Call syntax[141]==

```
Arguments:      [Argument // ','].
```
This macro is invoked in definition 101.

Function type syntax[142]==

```
ParamTypes:     [ParamType // ','].
```
This macro is invoked in definition 134.

12.4 Other concrete productions

Concrete pointer syntax[143]==

```
Operand: 'nil'.
```
This macro is invoked in definition 126.

Concrete notations are stated for the comma separated sequence of type denoters. The specific precedence of the cast expression and its parentheses avoid a parsing conflict. *Concrete union syntax*[144]==

```
UnitedTypes: UnitedTypes ',' UnitedType.
UnitedTypes: UnitedType.
Operand: '<' TypeDenoter '>' Operand.
```
This macro is invoked in definition 89.