Journal of Universal Computer Science, vol. 3, no. 5 (1997), 443-503
submitted: 20/12/96, accepted: 20/5/97, appeared: 28/5/97 © Springer Pub. Co.

The Formal Specification of Oberon

Philipp W. Kutter
(Eidgenössische Technische Hochschule, Switzerland
kutter@tik.ee.ethz.ch)

Alfonso Pierantonio
(Università di L'Aquila, Italy
alfonso@univaq.it)

Abstract: This paper presents the formal specification of the programming language Oberon. Using Montages we give a description of syntax, static, and dynamic semantics of all constructs of the language. The specification is arranged in five refinement steps, each of them results in a working sub–language of Oberon. The compactness and readability of the specification make us believe that it can be used for a reference manual.

Key Words: abstract state machines, Oberon, Montages, programming languages specifications, reference manuals

Category: F.3.2, D.3.1, D3.3

1 Introduction

In this paper we present the formal specification of the programming language Oberon [7] using Montages taking advantage of existing ASM specifications of the dynamic semantics of imperative programming languages [1, 2, 6, 3]. A detailed presentation of Montages is given in [4].

The language specification is presented as a sequence of five sub-languages of Oberon ($\mathcal{O}_1$ (section 2), $\mathcal{O}_2$ (section 3), $\mathcal{O}_3$ (section 4.1), $\mathcal{O}_4$ (section 4.2), $\mathcal{O}_5$ (section 5)), each extending its predecessor with some new constructs. The last of them is complete Oberon. The first (section 2) features variables and designators of pointer and record types. The expressions, procedures, modules, and finally type extensions are smoothly introduced in the following refinement steps.

The material is presented in such a way that new constructs are specified either with a new Montage (having M labels) or as a renaming of an existing Montage. Typically a new construct induces a number of small refinements of already given Montages.

Such refinements are indicating with a label to which part of the preceding Montage they do concern, the grammar (G labels), static analysis (A labels), static semantics condition (C labels), or dynamic semantics (D labels).

2 Basic Concepts and Record Types

In this section we present a first sub-language of Oberon. In contrast to most programming language manuals we do not start with expressions, but with primitive and composite objects, their typing, creation, and manipulation.

The constructs of Oberon can be divided in statements, expressions, Designators, and Types. Modules and procedures can be used to encapsulate, reuse and extend these constructs. In the Montages approach one token of each construct is chosen as task that executes the dynamic semantics of the construct. These tasks are grouped in universes StatementTask, ExpressionTask, DesignatorTask, and TypeTask respectively. All designators and expressions are statically typed. The field

$$\text{StaticType: (ExpressionTask} \cup \text{DesignatorTask)} \rightarrow \text{TypeTask} \qquad (\mathcal{O}_1.\Sigma.1)$$

denotes the type of an expression or designator.

2.1 Storage

The storage is modeled by a collection of abstract objects. An object is either a simple object or a composite object that is composed of objects again.

$$\text{Object} = \text{Primitive} \cup \text{Composite}$$

A primitive object, or location, is an object of an unstructured type, and its information is stored in a dynamic field

$$\text{Value: Primitive} \rightarrow \text{(OberonValue} \cup \text{Composite)} \qquad (\mathcal{O}_1.\Sigma.2)$$

Composite objects are used to model instances of structured types. Instances of such types consist of a composite object together with the objects it is composed of. These objects build typically a small tree, with the composite object as root. The descendants in the tree are accessed by a binary function

$$\text{Field: Composite} \times \text{(String} \cup \text{Nat)} \rightarrow \text{Object} \qquad (\mathcal{O}_1.\Sigma.3)$$

whose second argument chooses a component by a name (e.g. records) or a number(e.g. arrays). In the following we show pointers and records, as examples.

A *pointer* is modeled by a primitive object. The possible values of a pointer are composite objects and the predefined element *Nil.* A newly allocated pointer is guaranteed to be initialized with *Nil.*

A *record* is a composite object that is composed of a set of named and typed fields. Depending on their type the fields are locations or composite objects, which are again composed of other objects. A record is thus the root of a tree of arbitrary complexity. The leafs of such a tree are locations and the values of these locations are called the *values of a record.*

In figure 1 we show an example record having three fields a, b, and c, where the c field is a record having two fields d, and e. All circles are objects, the record-root and the c-field are composite objects. Fields a, b, d, and e are locations, and have thus a value. The targets of the value arrows are all going in a box representing the values of the record. The Oberon type describing such a records is denoted as follows,

```
RECORD
  a, b:   T
  c:      RECORD
            d,e:   T
          END
END
```

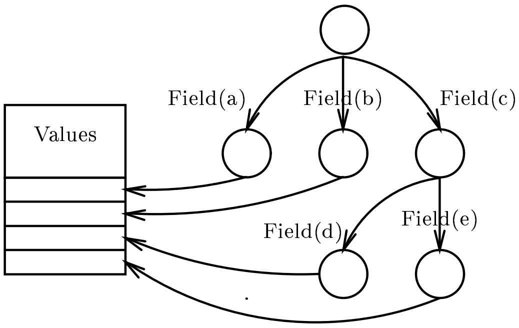


Figure 1: An example record and its values

where T is an arbitrary unstructured type.

Often one differentiates between the dynamic store, the stack, and store that is statically allocated within the program text. In our model there is only one store, the above introduced universe Objects. Tokens of the code which require storage, namely variables and parameters, allocate an object and establish a link to it through their field

$$\text{AssocOBJ: Token} \rightarrow \text{Object} \qquad (\mathcal{O}_1.\Sigma.4)$$

The variables and parameters are in turn referenced by designators.

The associated object field is used throughout the model to link designators to their semantically associated object in the store. The strong typing of Oberon guarantees that a variable, parameter, or designator with static type T is always associated with the root of an instance of T.

2.2 Class Type

A type T can be seen as an instance of a *class type* which provides services to create and copy objects of type T.

The parameters for the services are stored in four fields of the type, called *Mode*, *Return*, *Src* and *Dest*. A service is called by passing control to the type, and by setting the field mode to copy or create. The return field is used as return address to which control is passed after termination of a service.

The *copy service* sets the value(s) of a destination object to the value(s) of a source object. The two objects are provided as the *Src* and *Dest* fields of the type. In order to abstract from the technical details, we introduce a macro, which simulates a method call syntax. The macro

$$\begin{array}{ll} \textit{type}.\textbf{COPY}(\textit{source}, \textit{destination}) \quad \triangleq & (\mathcal{O}_1.\text{D}.1) \\ \quad \textit{type}.\text{Mode} := \text{copy} & \\ \quad \textit{type}.\text{Src} := \textit{source} & \\ \quad \textit{type}.\text{Dest} := \textit{destination} & \\ \quad \textit{type}.\text{Return} := \text{NextTask} & \\ \quad \text{CurrentTask} := \textit{type} & \end{array}$$

requires the copy service of *type* in the above described way: the mode field is set to copy, *source* and *destination* are assigned to the corresponding fields, the return field is set to the next task, and control is passed to *type*.

The *create service* creates a new instance of the type. In the case of structured types such an instance is a tree of objects; in the case of unstructured types it a single object. After termination of the service the root of the newly created instance is accessible as the *Dest* field of the type. The standard method to call a create service is given by the macro

type.**CREATE()** ; R $\triangleq$ ($\mathcal{O}_1$.D.2)
  if not Allocated **then**
    type.Mode := create
    type.Return := CurrentTask
    CurrentTask := *type*
    Allocated := true
  else
    Allocated := false
    R
  endif
  where NewObject $\triangleq$ *type*.Dest

In this macro we define the local macro **NewObject** which can be used in R to access the new object *type*.Dest. The semicolon ";" in the macro is used to indicate that transition rule R is executed after the allocation. The sequential execution of the steps is forced in the macro-definition using a boolean field Allocated. The rule R typically passes the control to the next task or calls a copy service in order to initialize the new object.

2.3 Variables

One of the main mechanism in programming languages is the declaration of named structures, and their use in the algorithms. Such structures include parts of the store, known as *variables*, code fragments, known as *procedures*, and properties of variables and procedures, so called *types*. In modern programming languages all three kinds of structures are created by declarations. A declaration consists typically of an identifier, denoting the *name*, and a *characterization*. Types are characterized by a construction from other types (e.g. record, pointer types), variables are characterized by types, and procedures are characterized by the type of their parameters, a code fragment using these parameters, and additional declarations.

In our model we call the identifiers in the declarations *code objects*.

CodeObject = Ident ($\mathcal{O}_1$.G.1)

The micro syntax of an Ident can be accessed as the attribute

Name: Ident $\rightarrow$ String ($\mathcal{O}_1.\Sigma.5$)

If a declaration is analyzed we update a dynamic function TABLE such that it maps the name to some distinguished token of the declaration.

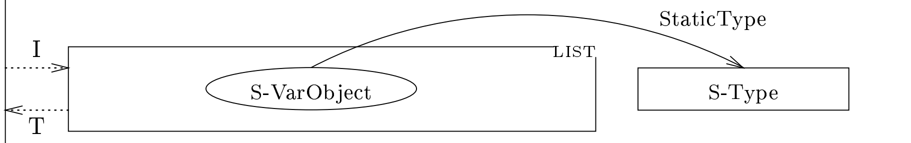
TABLE: String $\rightarrow$ Token ($\mathcal{O}_1.\Sigma.6$)

Typically the static analysis of references sets a field

Decl: Reference $\rightarrow$ Token ($\mathcal{O}_1.\Sigma.7$)

to the declaration. Like this, TABLE is only used during static analysis, in order to link references directly with their declaration. Later in the dynamic semantics, one can abstract from all static resolvable declaration and reference mechanisms.

VarDeclaration	**::=**	**VarObject{"," VarObject} ":" Type**
VarObject	**=**	**CodeObject**



vary v **over list** S-VarObject
  TABLE(v.Name) := v
endvary

condition **for all** v **in list** S-VarObject **holds**
  not Predefined(v.Name)

VarObject:

StaticType.**CREATE()** ;
  AssocOBJ:= **NewObject**
  CurrentTask := NextTask

Montage $\mathcal{O}_1$.M.1: The semantics of a variable declaration.

Montage $\mathcal{O}_1$.M.1 defines the syntax and semantic of variables and their declaration. Syntactically a variable declaration consists of a list of variable objects and one common characterization by a type. The field StaticType of all variable objects is set to the type descendant, and each variable object is put in the TABLE.

The static semantics guarantees only, that predefined names cannot be reused. The alert reader may wonder why there is no condition that the variable objects have different names. In fact, this condition is tested once for all declarations, e.g. variable, constant, type, and procedure declarations. (see declaration sequence Montage $\mathcal{O}_1$.M.14).

The dynamic semantics of a variable object calls the create service of its static type, and sets its associated object to the newly created object.

A simple designator (Montage $\mathcal{O}_1$.M.2) is used to reference a variable. The entry TABLE(Name) denotes the variable object of the declaration corresponding to the micro syntax of the simple. The static semantics guarantees the existence of this entry and prevents it from being a type. The static analysis sets Decl to the entry and the static type to the static type of the entry.

The dynamic semantics sets the associated object to the associated object of its declaration and passes control to the next task. The macro **SetTo** is defined as follows:

$$\mathbf{SetTo}(o) \quad \triangleq \quad \text{AssocOBJ:=}\ o \qquad (\mathcal{O}_1.\text{D.3})$$

Simple	**=**	**Ident**
	Decl := TABLE(Name) StaticType := TABLE(Name).StaticType	
condition	TABLE(Name) $\neq$ undef **and not** TypeTask(TABLE(Name))	
Simple:	SetTo(Decl.AssocOBJ) CurrentTask := NextTask	

Montage $\mathcal{O}_1$.M.2: The semantics of a simple designator.

2.4 Statements

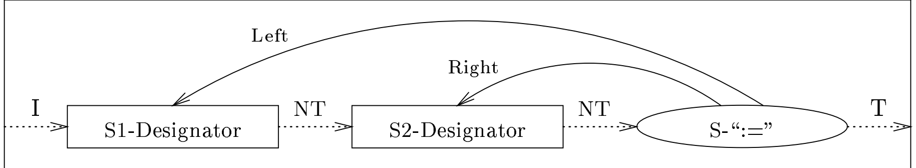
$$\text{StatementSequence} \quad ::= \quad \text{Statement } \{\text{“;” Statement}\} \qquad (\mathcal{O}_1.\text{G}.2)$$

The statements in a statement sequence are sequentially linked through the next task function in their syntactical order. This is done implicitly (see ListNode definitions in [4]). In this section we present the assignment and the new statement.

$$\text{Statement} \quad = \quad \text{Assignment} \mid \text{NewStatement} \qquad (\mathcal{O}_1.\text{G}.3)$$

The assignment (Montage $\mathcal{O}_1$.M.3) copies the value(s) of the associated object of the right designator to the value(s) of the associated object of the left designator. This is done using the copy service of the type of the left designator.

Assignment ::= **Designator “:=” Designator**



condition S2-Designator.StaticType AssignableTo S1-Designator.StaticType

“:=”:

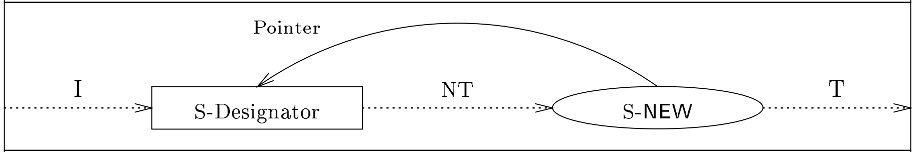
Left.StaticType.**COPY**(Right.AssocOBJ, Left.AssocOBJ)

Montage $\mathcal{O}_1$.M.3: The semantics of an assignment

The static semantics condition guarantees that the source and the destination have compatible types. It uses the macro AssignableTo, which states, that one can assign only objects of the same type or objects of pointers referencing to assignable types.

$$t1 \text{ AssignableTo } t2 \triangleq \quad (\mathcal{O}_1.\text{C.1})$$
$$t1 \neq \text{undef } \textbf{and } t2 \neq \text{undef } \textbf{and}$$
$$(t1 = t2$$
$$\textbf{or } t1.\text{ReferencedType AssignableTo } t2.\text{ReferencedType})$$

NewStatement ::= **NEW"(" Designator ")"**



condition POINTER(S-Designator.StaticType)

NEW:

Pointer.StaticType.PointedType.**CREATE()** ;
Pointer.AssocOBJ.Value := **NewObj**
CurrentTask := NextTask

Montage $\mathcal{O}_1$.M.4: The semantics of a new-statement.

The new statement (Montage $\mathcal{O}_1$.M.4) has a designator as argument. This designator is guaranteed to be of pointer type. The task of the NEW-token is to allocate a new record and to assign it to the value of the associated object of the designator. In order to understand why the type of the new record is Pointer.StaticType.PointedType one has to look at the pointer type Montage ($\mathcal{O}_1$.M.7).

In the last section we introduced variables which are associated with an object in the store. Simple designators can be used to refer to such an object. The above introduced assignment and the new statement are used to manipulate these objects and to allocate new objects. Their full power is only visible after we introduced concrete types, and complex designators. This will be done in the next two sections.

2.5 Types

A type declaration, as specified by the Montage $\mathcal{O}_1$.M.5, works like a variable declaration. The TABLE is used to link the declared type-names to the types.

In this section we introduce type identifiers, pointer types, and record types.

Type	=	TypeIdent | PointerType | RecordType	($\mathcal{O}_1$.G.4)
TypeIdent	=	Ident	

TypeDeclartion	**::=**	**TypeObject "=" Type**
TypeObject	**=**	**Ident**
Type	**=**	**TypeIdent** | **PointerType** | **PointableType**

S-TypeObject   S-Type

TABLE(S-TypeObject.Name) := S-Type

condition **not** Predefined(S-TypeObject.Name)

Montage $\mathcal{O}_1$.M.5: The semantics of a type declaration.

Type identifiers (Montage $\mathcal{O}_1$.M.6) serve as references to user defined types or to predefined basic types. The referenced type is denoted by TABLE(Name) and the static semantics guarantees that this entry is a type task. For reference purposes, the following definition lists all possibilities for type tasks. Most of them relate to universes that will be introduced in later sections.

$$\text{TypeTask} = \text{GroundType} \cup \mathsf{POINTER} \cup \mathsf{PROCEDURE} \cup \text{PointableTypeTask} \qquad (\mathcal{O}_1.\Sigma.8)$$

terminal leaves of a type identifier to the referenced type task. Like this, references to the type identifier are directly linked to the referenced type.

TypeIdent	**=**	**Ident**

Initial := TABLE(Name)
Terminal := TABLE(Name)

condition TypeTask(TABLE(Name))

Montage $\mathcal{O}_1$.M.6: Semantics of a type identifier

A pointer type consists of the keywords POINTER OF and the pointed type. An instance of a pointer type is guaranteed to point either to *NIL* or to an instance of the pointed type. The pointable types for this sub-language are record types.

$$\text{PointableType} \quad = \quad \text{RecordType} \qquad (\mathcal{O}_1.\text{G}.5)$$

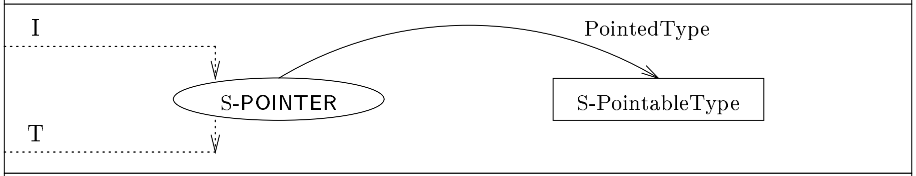
and the pointable type tasks are the RECORD-tokens:

$$\text{PointableTypeTask} = \mathsf{RECORD} \qquad (\mathcal{O}_1.\text{C}.2)$$

The declaration of pointer types allows to use a not yet declared record type. Such a pointer type must be of the form POINTER TO Ident, where the identifier is the name of a possibly undeclared record type. This form of pointer type is generated by the non-terminal PointerToIdent, whereas the other pointer types are generated by the non-terminal PointerToAnonym.

$$\text{PointerType} \quad = \quad \text{PointerToIdent} \mid \text{PointerToAnonym} \qquad (\mathcal{O}_1.\text{G}.6)$$

PointerToAnonym ::= **POINTER TO PointableType**



condition PointableTypeTask(S-PointableType)

POINTER:

```
if Mode = copy then
   Dest.Value := Src.Value
endif
CurrentTask := Return
```

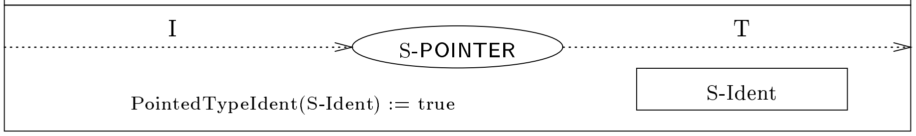
```
if Mode = create then
   extend Primitive with o
      Dest := o
      o.Value := Nil
   endextend
endif
CurrentTask := Return
```

Montage $\mathcal{O}_1$.M.7: Semantics of a pointer to an anonymous type.

Pointer types to anonymous types (Montage $\mathcal{O}_1$.M.7) can check their static semantics and perform their static analysis directly. The field PointedType denotes the type to which the pointer "points".

The dynamic semantics of a pointer type applies independently whether the type was generated by PointerToAnonymType or PointerToIdent. The rule has two parts which define the create and copy services. The create service allocates a new location and the value of the new location is initialized with Nil, whereas the copy service copies the value field of the source to the value field of the destination.

PointerToIdent ::= **POINTER TO Ident**

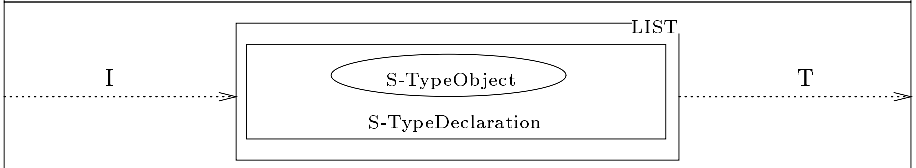


Montage $\mathcal{O}_1$.M.8: The semantics of a pointer to identifier type.

The Montage $\mathcal{O}_1$.M.8 shows the semantics of a pointer to identifier. The static analysis includes the identifier in the universe PointedTypeIdent.

The type declaration sequence (Montage $\mathcal{O}_1$.M.9) finally does the static analysis and checks the static semantics for all identifiers which have been collected in the PointedTypeIdent set. The static semantics condition is, that their entry in the TABLE is a pointable type token, and the static analysis sets the PointedType field to this type token. The universe PointedTypeIdent is emptied, in order to reuse it later.

TypeDeclSequence ::= [**TYPE** {**TypeDeclaration** ";"}]



vary p **over** PointedTypeIdent
  p.S-POINTER.PointedType := TABLE(p.S-Ident.Name)
  PointedTypeIdent(p) := false
endvary

condition **for all** p: PointedTypeIdent **holds**
  PointableTypeTask(TABLE(p.S-Ident.Name))

Montage $\mathcal{O}_1$.M.9: Semantics of a type declaration sequence.

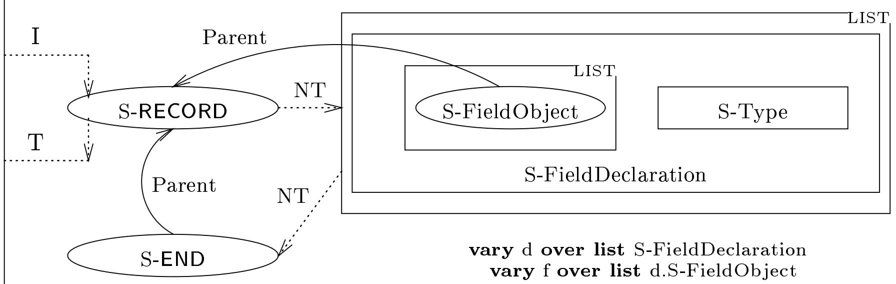
A record type (Montage $\mathcal{O}_1$.M.10) consists of a list of field declarations. These declarations define *fields* of the record type. The static semantics condition of RecordType guarantees that all field objects have different names. We use the following macro, which checks a *condition* for all fields f1 of *record1* and all fields f2 of *record2*.

ForFieldObjects f1 Of *record1* ($\mathcal{O}_1$.C.3)
AndForFieldObjects f2 Of *record2*
Holds *condition* $\triangleq$
  for all d1 **in list** *record1*.S-FieldDeclaration **holds**
    for all f1 **in list** d1.S-FieldObject **holds**
      for all d2 **in list** *record2*.S-FieldDeclaration **holds**
        for all f2 **in list** d2.S-FieldObject **holds**
          condition

The static analysis of a record type sets the attribute Parent of each field object to the RECORD-token, and defines the control-flow such that sequentially the RECORD-token, then all field objects and at the end the END-token get the control. In addition the function Field is used to link the RECORD-token with the field objects. Like this the structure of records and record types is unified (see section 2.1).

The create service of a record type allocates a record r, then each field object f_i creates an object o_i using the the create service of its type, and then the corresponding field $f_i.Name$ of r is set to o_i. After all field objects had the control, the control is passed to the END-token that passes the control to the return address of the record type. Thus the task of the *RECORD*-token is only to create an new composite object, to assign it to its Dest field, and to pass control to the first field. The create service of the field object (see Montage $\mathcal{O}_1$.M.11) is then calling the create service of its static type, and assigns the resulting new object to the corresponding field of the parent–object.

```
RecordType    ::=  RECORD
                   [FieldDeclaration {";" FieldDeclaration}] RecordEnd
RecordEnd     =    END
```




```
vary d over list S-FieldDeclaration
  vary f over list d.S-FieldObject
    S-RECORD.Field(f.Name) := f
  endvary
endvary
```

```
condition  ForFieldObjects f1 Of Self
           AndForFieldObjects f2 Of Self
           Holds      f1≠ f2 implies f1.Name ≠ f2.Name
```

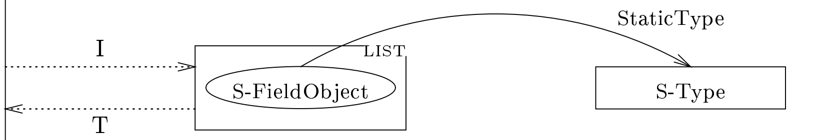
```
RECORD:
           if Mode = copy then
             CurrentTask := NextTask
           endif

if Mode = create then
  extend Composite with o
    Dest := o
    CurrentTask := NextTask
  endextend
endif

RecordEnd:
           CurrentTask := Parent.Return
```

Montage $\mathcal{O}_1$.M.10: The semantics of a record type.

```
FieldDeclaration  ::=  FieldObject {"," FieldObject} ":" Type
FieldObject       =    CodeObject
```




```
FieldObject:
  if Mode = create then
    StaticType.CREATE() ;
      Parent.Dest.Field(Name) := NewObject
      CurrentTast := NextTask
  endif

  if Mode = copy then
    StaticType.COPY(
             Parent.Src.Field(Name),
             Parent.Dest.Field(Name))
  endif
```

Montage $\mathcal{O}_1$.M.11: The semantics of a field declaration.

The copy service of a record type passes control directly to the field objects which copy the values of the corresponding fields by calling the copy services of their types. Both services terminate if control is passed to the RecordEnd whose dynamic semantics passes control to the return address.

2.6 Designators

Designators are used to reference objects in the store. We know already the simple designator ($\mathcal{O}_1$.M.2), which references the associated object of a variable. As seen there, the dynamic semantics of a designator task calculates the referenced object and assigns it to its AssocOBJ-field. For reference purposes we give the full definition of the universe

$$\begin{aligned}\text{DesignatorTask} = {} & \text{Simple} \cup \text{“}\uparrow\text{”} \cup \text{FieldSelector} \\ & \text{IndexTask} \cup \text{ShIndexTask} \cup \text{Guard}\end{aligned} \qquad (\mathcal{O}_1.\Sigma.9)$$

In this section we introduce dereferenced and qualified designators.

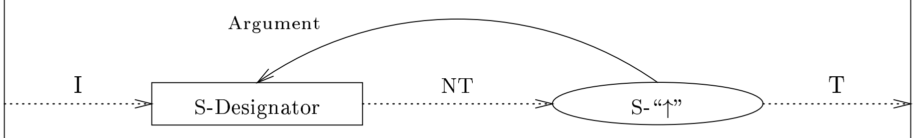
$$\text{Designator} \quad = \quad \text{Simple} \mid \text{Dereferenced} \mid \text{Qualified} \qquad (\mathcal{O}_1.\text{G}.7)$$

Both of them are constructions having as argument another designator. For the definitions we use the macro

$$\text{ArgType} \quad \triangleq \quad \text{S-Designator.StaticType} \qquad (\mathcal{O}_1.\text{C}.4)$$

which denotes the static type of this argument.

Dereferenced ::= **Designator “↑”**



S-“↑”.StaticType := ArgType.PointedType

condition POINTER(ArgType)

“↑”:

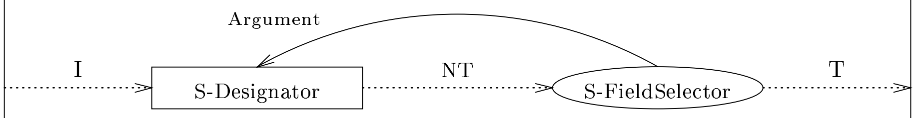
```
if Argument.AssocOBJ.Value ≠ Nil then
   AssocOBJ:= Argument.AssocOBJ.Value
   CurrentTask := NextTask
else
   CurrentTask := RunTimeError
endif
```

Montage $\mathcal{O}_1$.M.12: The semantics of a dereferenced designator

The dereferenced designator (Montage $\mathcal{O}_1$.M.12) is used to construct from a reference to a pointer p, a referenced to the pointed object p.Value. If the pointed object is *Nil*, a runtime error is raised. The static semantics condition of a dereferenced designator is that the argument is a pointer.

The qualified designator (Montage $\mathcal{O}_1$.M.13) is used to access fields of records. The argument designator may reference either a record or a pointer. In the first case we include the field selector in the universe DirectSelector and in the second case we include it in the universe IndirectSelector.

Qualified ::= **Designator "." FieldSelector**
FieldSelector = **Ident**




```
if RECORD(ArgType) then
   S-FieldSelector.StaticType := DirectField.StaticType
   DirectSelector(S-FieldSelector) := true
elseif POINTER(ArgType) then
   S-FieldSelector.StaticType := IndirectField.StaticType
   IndirectSelector(S-FieldSelector) := true
endif
```

condition RECORD(ArgType) **and** DirectField $\neq$ undef
or (POINTER(ArgType) **and** IndirectField $\neq$ undef)

DirectFieldSelector:

```
AssocOBJ:= Field(Argument.AssocOBJ, Name)
CurrentTask := NextTask
```

IndirectFieldSelector:

```
if Argument.AssocOBJ.Value ≠ Nil then
   AssocOBJ:= Field(Argument.AssocOBJ.Value, Name)
   CurrentTask := NextTask
else
   CurrentTask := RunTimeError
endif
```

Montage $\mathcal{O}_1$.M.13: The semantics of a qualified designator

As well depending on these cases, the selected field of the record type is either directly accessible

$$\text{DirectField} \triangleq \text{ArgType.Field(S-FieldSelector.Name)} \qquad (\mathcal{O}_1.\text{A.1})$$

or accessible via implicit dereferencing:

$$\text{IndirectField} \triangleq \text{ArgType.PointedType.Field(S-FieldSelector.Name)} \qquad (\mathcal{O}_1.\text{A.2})$$

The static semantics guarantees that in the case of a record argument-type the direct field is defined and in the case of a pointer argument-type the indirect field is defined.

The dynamic semantics of a direct field selector sets the associated object to the corresponding field of the argument. In the case of an indirect field selector, there is the possibility that the associated object of the argument is a pointer to Nil. This case leads to a run-time error, and control is passed to RunTimeError.

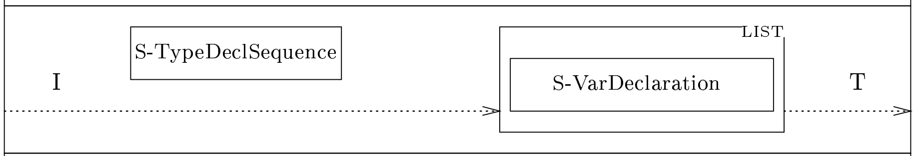
Otherwise the argument is dereferenced and the associated object is set to the corresponding record-field.

Remark: In the static analysis we use the Field-function to access fields of the record type, and in the dynamic semantics we use the same function Field to access the fields of a record object.

2.7 Programs

Up to now we defined a core of an imperative language with pointers. Here we complete it to a working sub-language.

DclSequence ::= **TypeDeclSequence [VAR {VarDeclaration ";"}]**



condition **for all** i1, i2: CodeObject
satisfying i1 CodeObjectOf Self **and** i2 CodeObjectOf Self
holds i1 $\neq$ i2 **implies** i1.Name $\neq$ i2.Name

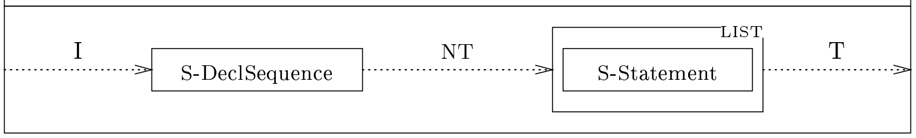
Montage $\mathcal{O}_1$.M.14: Semantics of a declaration sequence

A declaration sequence (Montage $\mathcal{O}_1$.M.14) contains all declarations. Its static semantics guarantees that the code objects have different names. To define this predicate we need to know all code objects of a declaration sequence.

i CodeObjectOf ds $\triangleq$ ($\mathcal{O}_1$.C.5)
exists vd **in list** ds.S-VarDeclaration
such that i **in list** vd.S-VarObject
or
exists td **in list** ds.S-TypeDeclSequence.S-TypeDeclaration
such that i = td.S-TypeObject

The construct Program (Montage $\mathcal{O}_1$.M.15) is not existing in Oberon, but we use it until we introduce modules in section 5. Until then the non-terminal Program is the start symbol of our grammar.

Program ::= **PROGRAM DeclSequence**
BEGIN Statement {";" Statement} END



Montage $\mathcal{O}_1$.M.15: The semantics of a program (not existing in Oberon).

3 Adding Expressions and Related Concepts

The sub-language given in the last section missed expressions and their evaluation. The result of the evaluation of an expression is stored as a field:

$$\text{VALUE: ExpressionTask} \rightarrow (\text{OberonValue} \cup \text{Composite}) \qquad (\mathcal{O}_2.\Sigma.1)$$

where OberonValue is the universe of all possible values in Oberon, e.g. numbers, characters, strings, booleans, and *Nil*. For reference purposes we give here the definition of ExpressionTask, most involved universes will be explained later in this section.

$$\begin{aligned}\text{ExpressionTask} = {} & \text{Constant} \cup \text{DesignatorTask} \cup \text{“}{\sim}\text{”} \cup \text{Sign} \\ & \cup \text{RelOp} \cup \text{AddOp} \cup \text{MulOp} \cup \text{“\&”} \cup \text{“OR”}\end{aligned} \qquad (\mathcal{O}_2.\Sigma.2)$$

3.1 Constants

Constant	= **number** | **character** | **string** | **set** | **TRUE** | **FALSE** | **NIL**
	StaticType := LeastContainingGroundType(ConstValue)
condition	LeastContainingGroundType(ConstValue) $\neq$ undef
Constant:	VALUE := ConstValue CurrentTask := NextTask

Montage $\mathcal{O}_2$.M.1: The semantics of a constant.

The most basic expressions are constants (Montage $\mathcal{O}_2$.M.1), which evaluate always to the same result. For simplicity we assume that their micro-syntax is already transformed to the corresponding semantical entity (a number, a string, a character, a set, or one of the elements false, true, or Nil) and accessible as the field

$$\text{ConstValue: Constant} \rightarrow \text{OberonValue} \qquad (\mathcal{O}_2.\Sigma.3)$$

For all expressions consisting only of constants the field ConstValue is calculated during static analysis. We will comment on this static calculations in the section 3.5.

The dynamic semantics of a constant simply copies its ConstValue into the VALUE field and passes control to the next task. The static type of a constant is the least containing ground type of its constant value. The static function

$$\text{LeastContainingGroundType: OberonValue} \rightarrow \text{GroundType} \qquad (\mathcal{O}_2.\Sigma.4)$$

maps a value to the least of the ground types which contains the value. If none of the ground types contains the value, the result is undef. The predefined ground types are:

$$\begin{aligned}&\text{GroundType} = \text{NumericType} \cup \\ &\qquad \{\mathsf{BOOLEAN}, \mathsf{CHAR}, \mathsf{STRING}, \mathsf{SET}\}\\ &\text{NumericType} = \text{IntegerType} \cup \{\mathsf{REAL}, \mathsf{LONGREAL}\}\\ &\text{IntegerType} = \{\mathsf{SHORTINT}, \mathsf{INTEGER}, \mathsf{LONGINT}\}\end{aligned} \qquad (\mathcal{O}_2.\Sigma.5)$$

Here we only need to know, that numeric types have two static fields,

$$\begin{aligned}&\text{Min: } (\text{NumericType} \cup \{\mathsf{SET}\}) \rightarrow \text{OberonValue}\\ &\text{Max: NumericType} \rightarrow \text{OberonValue}\end{aligned} \qquad (\mathcal{O}_2.\Sigma.6)$$

which deliver their smallest and the largest permitted value. Max(SET) is the largest integer that may be put in an Oberon set. In order to integrate ground types nicely in our model, we initialize the TABLE with links to them. TABLE thus maps the string "SHORTINT" to SHORTINT, e.t.c. The dynamic semantics of the ground types can now be given as follows:

($\mathcal{O}_2$.D.1)

GroundType:

```
if Mode = create then
  extend Primitive with o
    Dest := o
    CurrentTask := Return
  endextend
elseif  Mode = copy then
  if NumericType(Self) and
    (Scr.VALUE < Min or Scr.VALUE > Max) then
      CurrentTask := RunTimeError
  else
    Dest.VALUE := Src.VALUE
    CurrentTask := Return
  endif
endif
```

3.2 Factors

Composed expressions are built up from factors. Besides constants, factors may be designators, negated factors, or expressions in parentheses.

Factor = Constant | Designator | NegatedFactor | ExprInParentheses ($\mathcal{O}_2$.G.1)

Designators may be used as expressions only if they are of unstructured type. In this case a designator task must set not only its AssocOBJ field, but as well its VALUE field. In order to achieve this, we have to redefine the definition of the macro SetTo ($\mathcal{O}_1$.D.3) as follows

SetTo(o) $\triangleq$ ($\mathcal{O}_2$.D.2)

```
  AssocOBJ:= o
  if UnstructuredType(StaticType) then
    VALUE := o.Value
  endif
```

where UnstructuredType is the union of GroundType and POINTER. After this redefinition all designator specifications of the last section can be reused.

In the first sub-language (2) we presented assignments with two designator arguments ($\mathcal{O}_1$.M.3). Here we generalize them to assignments from expressions to designators, by changing the grammar to

Assignment ::= Designator ":=" Expression ($\mathcal{O}_2$.G.2)

and by substituting S1-Designator and S2-Designator with S-Designator and S-Expression respectively. Like this the static parts of Montage $\mathcal{O}_1$.M.3 remain valid, but we have to redefine the macro AssignableTo (see $\mathcal{O}_1$.C.1 for the old definition) such that it allows to assign an object of some ground type to another one including this type.

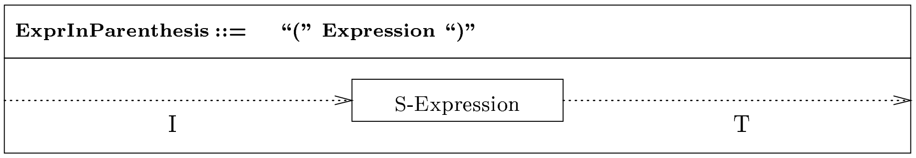
t1 AssignableTo *t2* $\triangleq$ ($\mathcal{O}_2$.C.1)
  t1 $\neq$ undef **and** *t2* $\neq$ undef **and**
   (*t1* = *t2*
   or *t1*.ReferencedType AssignableTo *t2*.ReferencedType
   or *t1* $\preccurlyeq$ *t2*)

SHORTINT $\preccurlyeq$ INTEGER $\preccurlyeq$ LONGINT $\preccurlyeq$ REAL $\preccurlyeq$ LONGREAL ($\mathcal{O}_2$.C.2)

The dynamic semantics of an Assignment must be refined as well. The right hand side is now an expression, which possibly has only a value, but no associated object. Thus in the case of unstructured types we have to assign directly the value of the right-hand-side to the value of the associated object of the left-hand-side.

":=": ($\mathcal{O}_2$.D.3)
  if UnstructuredType(Left.StaticType) **then**
   Left.AssocOBJ.Value := Right.VALUE
   CurrentTask := NextTask
  else
   Left.StaticType.**COPY**(Right.AssocOBJ, Left.AssocOBJ)
  endif

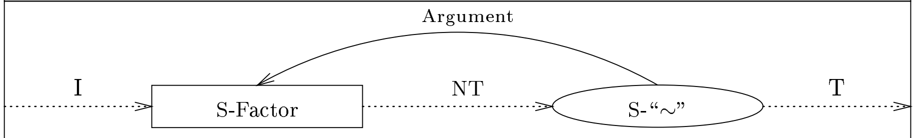
The only purpose of the ExprInParentheses Montage ($\mathcal{O}_2$.M.2) is to define the initial and terminal leaves of this production.



Montage $\mathcal{O}_2$.M.2: Semantics of an expression in parenthesis.

A negated factor (Montage $\mathcal{O}_2$.M.3) must have an argument of boolean type; the static type of the "~"-task is set to BOOLEAN and the constant value is set to the negated constant value of the argument [1]; the dynamic semantics sets the value to the negated value of the argument.

NegatedFactor ::= "∼" Factor



S-"∼".StaticType := BOOLEAN
S-"∼".ConstValue := ¬ S-Factor.ConstValue

condition S-Factor.StaticType = BOOLEAN

"∼":

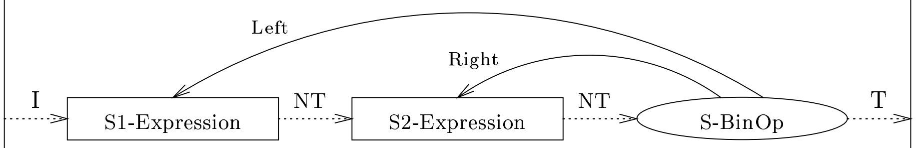
VALUE := ¬ Argument.VALUE
CurrentTask := NextTask

Montage $\mathcal{O}_2$.M.3: Semantics of a negated factor

3.3 Composed Expressions

The binary expression Montage $\mathcal{O}_2$.M.4 is an abstract Montage which will be refined to concrete Montages later. Its static analysis sets the data-flow functions Left and Right, calculates the type, and the constant value.

BinaryExpression ::= Expression BinOp Expression



S-BinOp.StaticType := CalculatedType
S-BinOp.ConstValue := ConstResult

condition CalculatedType ≠ undef **and** ConstCalculationsCondition

Montage $\mathcal{O}_2$.M.4: Generic Montage for binary expressions.

The calculated type is

$$\text{CalculatedType} \triangleq \text{ResultingType(S-BinOp, S1-Expression.StaticType, S2-Expression.StaticType)} \qquad (\mathcal{O}_2.\text{A}.1)$$

and the function ResultingType is defined as follows:

[1] In our setting the negation of undef is undef

Operator	Left Type	Right Type	Result Type
SymmetricRelOp	l ∈ NumericType	r ∈ NumericType	BOOLEAN
SymmetricRelOp	CHAR	CHAR	BOOLEAN
SymmetricRelOp	STRING	STRING	BOOLEAN
"=", "#"	SET	SET	BOOLEAN
"=", "#"	l ∈ POINTER	r ∈ POINTER: l AssignableTo r **or** r AssignableTo l	BOOLEAN
"=", "#"	l ∈ PROCEDURE	r ∈ PROCEDURE: IdenticProcTypes(l, r)	BOOLEAN
"=", "#", OR, "&"	BOOLEAN	BOOLEAN	BOOLEAN
"+", "-", "*"	l ∈ NumericType	r ∈ NumericType	LCST(l,r)
"/"	l ∈ NumericType	r ∈ NumericType	LCST(REAL, LCST(l,r))
DIV, MOD	l ∈ IntegerType	r ∈ IntegerType	LCST(l,r)
"+", "-", "*", "/"	SET	SET	SET
IN	l ∈ IntegerType	SET	BOOLEAN

The definition of LeastCommonSuperType (in the table abbreviated to LCST) is defined over ground types according to the sub-typing $\mathcal{O}_2$.C.2.

The binary expressions Product, Term, Sum, and Relation are all refinements of the Montage $\mathcal{O}_2$.M.4. The parts of Montage $\mathcal{O}_2$.M.4 are reused by replacing S1-Expression, and S2-Expression with the selector functions for the left and right argument and by replacing S-BinOp with the selector function for the operator. Taking for instance

Sum	::=	SimpleExpression AddOp Term
AddOp	=	"+" | "-"

we get the corresponding Montage by taking Montage $\mathcal{O}_2$.M.4 and replacing S1-Expression with S-SimpleExpression, S2-Expression with S-Term, and S-BinOp with S-AddOp. In summary the grammar rules for composed expressions are:

Expression	=	SimpleExpression | Relation	($\mathcal{O}_2$.G.3)
Relation	::=	SimpleExpression RelOp SimpleExpression	
RelOp	=	SymmetricRelOp | IN	
SymmetricRelOp	=	"=" | "#" | "<" | "<=" | ">" | ">="	
SimpleExpression	=	Term | SignedTerm | Sum | Disjunction	

SignedTerm	::=	Sign Term	($\mathcal{O}_2$.G.4)
Sign	=	"+" | "-"	
Sum	::=	SimpleExpression AddOp Term	
AddOp	=	"+" | "-"	
Term	=	Factor | Product | Conjunction	
Product	::=	Term MulOp Factor	
MulOp	=	"*" | "/" | DIV | MOD	

For the dynamic semantics rules of the operators we use a function

$$\text{Apply: (RelOp} \cup \text{AddOp} \cup \text{MulOp)} \times \text{OberonValue} \times \text{OberonValue} \rightarrow \text{OberonValue} \qquad (\mathcal{O}_2.\Sigma.7)$$

which applies the operators to two values. The definition of Apply corresponds to the usual mathematical meaning of the symbols. More precisely, the numeric symbols are interpreted in a standard way, DIV is interpreted as the integer part of the result of a division, MOD as the modulo function, and IN is the $\in$ relation. If one of the arguments is undefined, the result is undefined as well. For convenience we define the macro

$$\text{Result} \triangleq \text{Apply(CT, Left.VALUE, Right.VALUE)} \qquad (\mathcal{O}_2.\text{D}.4)$$

which produces the value of the binary expression. For the calculation of the constant value we use a macro

$$\text{ConstResult} \triangleq \text{Apply(S-BinOp, S1-Expression.ConstValue, S2-Expression.ConstValue)} \qquad (\mathcal{O}_2.\text{A}.2)$$

which produces the constant value, if for both arguments the constant value is defined. In the following we give the dynamic semantics of the different operators. The definition of the macro ConstCalculationsCondition ($\mathcal{O}_2$.C.3) reflects the runtime tests in the static semantics and is given at the end.

The dynamic semantics of relation operations and of MOD just sets the value and passes control to the next task.

RelOp, MOD: ($\mathcal{O}_2$.D.5)
 VALUE := Result
 CurrentTask := NextTask

The dynamic semantics of numeric operators can raise run–time–errors on several occasions. First of all their may be overflows. A static boolean macro

$$v \text{ ContainedIn } t \triangleq \text{v} \leq \text{t.Min } \mathbf{and} \text{ v} \leq \text{t.Max} \qquad (\mathcal{O}_2.\text{D}.6)$$

tests whether a number is contained in a numeric type of Oberon. Using this macro the dynamic semantics for binary addition, subtraction, and multiplication raises a run-time-error if there is an overflow.

AddOp, "∗": ($\mathcal{O}_2$.D.7)
if Result ContainedIn StaticType **then**
VALUE := Result
CurrentTask := NextTask
else
CurrentTask := RunTimeError
endif

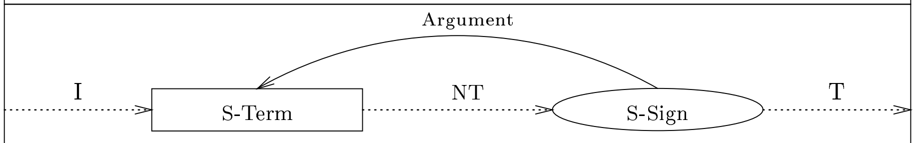
The dynamic semantics of the two division operators "/" and DIV raises in addition a run-time-error if the right argument is 0.

"/", DIV: ($\mathcal{O}_2$.D.8)
if Result ContainedIn StaticType **and**
Right.VALUE $\neq$ 0 **then**
VALUE := Result
CurrentTask := NextTask
else
CurrentTask := RunTimeError
endif

The static semantics checks which have to be done for the constant calculations correspond to the above run time checks. They are summarized in the macro

ConstCalculationsCondition $\triangleq$ ($\mathcal{O}_2$.C.3)
ConstResult $\neq$ undef **implies**
(NumericType(CalculatedType) **implies**
ConstResult ContainedIn CalculatedType)
and (("/"(S-BinOp) **or** DIV(S-BinOp)) **implies**
S2-Expression.ConstValue $\neq$ 0)

SignedTerm ::= Sign Term



S-Sign.StaticType := S-Term.StaticType
S-Sign.ConstValue := Apply(S-Sign, 0, S-Term.ConstValue)

condition NumericType(S-Term.StaticType) **and**
(S-Term.ConstValue $\neq$ undef **implies**
Apply(S-Sigh, 0, S-Term.ConstValue) ContainedIn S-Term.StaticType)

Sign:
if Apply(CT, 0, Argument.VALUE) ContainedIn StaticType **then**
VALUE := Apply(CT, 0, Argument.VALUE)
CurrentTask := NextTask
else
CurrentTask := RunTimeError
endif

Montage $\mathcal{O}_2$.M.5: The semantics of a signed term

The range check in the signed term Montage ($\mathcal{O}_2$.M.5) is needed since ranges of ground types may be asymmetric. The constant calculations again reflect the dynamic case.

3.4 Conjunction and Disjunction

The logical expressions Conjunction and Disjunction are lazy evaluated. Both of them are refinements of the Montage $\mathcal{O}_2$.M.6. The abstract Montage is again reused by replacing S1-Expression, and S2-Expression with the selector functions for the left and right argument and by replacing S-LogOp with the selector function for the corresponding operator.

Conjunction	::=	Term "&" Factor	($\mathcal{O}_2$.G.5)
Disjunction	::=	SimpleExpression OR Term	

The dynamic semantics rules for both productions use a flag

RightVisited: ("&" $\cup$ OR) $\rightarrow$ Boolean ($\mathcal{O}_2.\Sigma$.8)

which is used to remember whether only the left argument has been evaluated or as well the right one. If the result of the logical expression can be deduced only from the left argument, the right argument is never getting control.

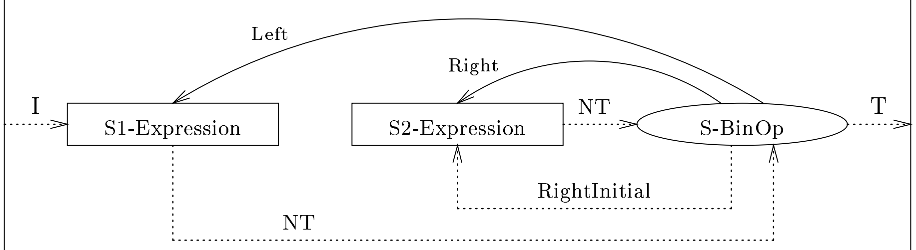
"&": ($\mathcal{O}_2$.D.9)

```
if Left.VALUE = true then
  if RightVisited then
    VALUE := Right.VALUE
    RightVisited := false
    CurrentTask := NextTask
  else
    RightVisited := true
    CurrentTask := RightInitial
  endif
else
  VALUE := false
  CurrentTask := NextTask
endif
```

OR: ($\mathcal{O}_2$.D.10)

```
if Left.VALUE = false then
  if RightVisited then
    VALUE := Right.VALUE
    RightVisited := false
    CurrentTask := NextTask
  else
    RightVisited := true
    CurrentTask := RightInitial
  endif
else
  VALUE := true
  CurrentTask := NextTask
endif
```

LogicalExpression ::= Expression LogOp Expression



S-LogOp.StaticType := BOOLEAN
S-LogOp.ConstValue :=
Apply(S-LogOp, S1-Expression.ConstValue, S2-Expression.ConstValue)

condition Left.StaticType = BOOLEAN **and** Right.StaticType = BOOLEAN

Montage $\mathcal{O}_2$.M.6: The semantics of logical expressions

3.5 Constant Declarations

A constant declaration binds an identifier to an expression. This expression must be constant.

In summary the mechanism for constant evaluation works as follows. Each constant and constant object sets its constant value to the defined value. A simple designator copies this field, thus if it was designating a constant object, the field is set to a value, else to undef. Composite expressions apply their operator to the constant values of their argument. If one of the arguments was not a constant, e.g. its constant value is undef, the resulting constant value is undef as well. The static semantics of a constant declaration (Montage $\mathcal{O}_2$.M.7) guarantees that the ConstValue of the expression descendant is defined.

ConstDeclaration ::= ConstObject "=" Expression
ConstObject = CodeObject

TABLE(S-ConstObject.Name) := S-ConstObject
S-ConstObject.ConstValue := S-Expression.ConstValue
S-ConstObject.StaticType := S-Expression.StaticType

condition S-Expression.ConstValue $\neq$ undef **and**
not Predefined(S-ConstObject.Name)

Montage $\mathcal{O}_2$.M.7: The semantics of a constant declaration.

In the Montages $\mathcal{O}_2$.M.1, $\mathcal{O}_2$.M.3, $\mathcal{O}_2$.M.4, $\mathcal{O}_2$.M.5, and $\mathcal{O}_2$.M.6 the ConstValue is defined for all expressions except simple designators. For simple designators, we have to extend the static analysis rule of Montage $\mathcal{O}_1$.M.2 with the following rule:

BLOCK STATIC ANALYSIS OF SIMPLE WITH: ($\mathcal{O}_2$.A.3)
    if ConstObject(TABLE(Name)) **then**
      ConstValue := TABLE(Name).ConstValue
    endif

In order to use constants at run-time, we need to refine as well the dynamic semantics rule of simple designators (Montage $\mathcal{O}_1$.M.2).

Simple: ($\mathcal{O}_2$.D.11)
    if VarObject(Decl) **then**
      SetTo(Decl.AssocOBJ)
    elseif ConstObject(Decl) **then**
      VALUE := Decl.ConstValue
    endif
    CurrentTask := NextTask

What now remains to do is to add constant declarations to the declaration sequence, which is described in Montage $\mathcal{O}_1$.M.14. As a first step we prefix the right-hand-side of the DeclSequence production rule with

PREFIX R-H-S OF ($\mathcal{O}_2$.G.6)
DECLSEQUENCE PRODUCTION $\mathcal{O}_1$.M.14 WITH:
[CONST {ConstDeclaration ";"}]

Then we add to the graphical part of Montage $\mathcal{O}_1$.M.14 a list box containing a box labeled with S-ConstDeclaration. This nested pair of boxes is putted between the left border and the S-TypeDeclSequence box. This means that the constants are analyzed first, e.g. they can be used in the other declarations.

The static semantics of declaration sequences remains valid, but we have to refine the macro CodeObjectOf by disjuncting its old definition ($\mathcal{O}_1$.C.5) with

DISJUNCT DEFINITION OF ($\mathcal{O}_2$.C.4)
CODEOBJECTOF $\mathcal{O}_1$.C.5 WITH:
    exists cd **in list** ds.S-ConstDeclaration
      such that i = cd.S-ConstObject

3.6 Control Statements

The sub-language of section 2 allowed only a fixed sequence of statements to be executed. Control statements allow conditional execution, selection, or repetition of statement sequences.

Oberon features five control statements, which are added to the choices of the synonym production $\mathcal{O}_1$.G.3:

Statement	::=	Assignment | NewStatement | IfStatement | CaseStatement | WhileStatement | RepeatStatement | LoopStatement	($\mathcal{O}_2$.G.7)

The Montages $\mathcal{O}_2$.M.11, $\mathcal{O}_2$.M.12, $\mathcal{O}_2$.M.14, $\mathcal{O}_2$.M.13, $\mathcal{O}_2$.M.8, $\mathcal{O}_2$.M.9 are self explanatory. The loop Montage ($\mathcal{O}_2$.M.10) uses a global nullary function

GlobalBlockEnd: EndLoop ($\mathcal{O}_2.\Sigma$.9)

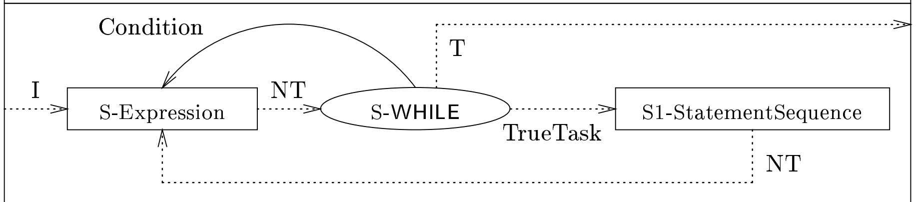
which is used to store the current block end within a loop. An exit statement (see Montage $\mathcal{O}_2$.M.10) passes control to that block end. If there is a nested loop within a loop, the old block end is stored as the field OldBlockEnd of the LOOP-task and GlobalBlockEnd is set to the end of the inner loop. At the end of the inner loop GlobalBlockEnd is set back to the old value.

The static semantics guarantees that there is no exit statement outside of a loop. This condition is part of the Montage of the start symbol Program.

NoExitOutsideLoop $\triangleq$ ($\mathcal{O}_2$.C.5)
  for all e **in** EXIT **holds**
    exists l **in** LoopStatement **such that**
      e SyntacticallyIn l

n1 SyntacticallyIn *n2* $\triangleq$ ($\mathcal{O}_2$.C.6)
  n1.Up = *n2* **or**
  n1.Up SyntacticallyIn *n2*

WhileStatement ::= **WHILE Expression DO StatementSequence END**



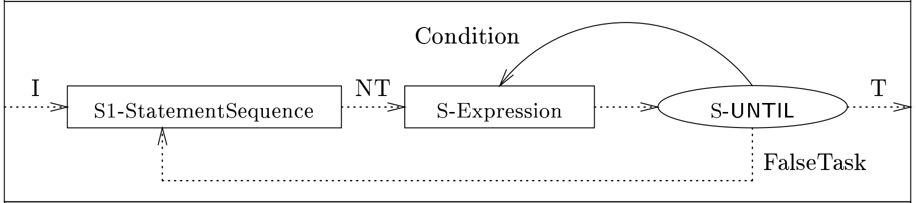
condition S-Expression.StaticType = BOOLEAN

WHILE:

```
if Condition.VALUE = true then
   CurrentTask := TrueTask
else
   CurrentTask := NextTask
endif
```

Montage $\mathcal{O}_2$.M.8: Semantics of a while statement

RepeatStatement ::= **REPEAT StatementSequence UNTIL Expression END**



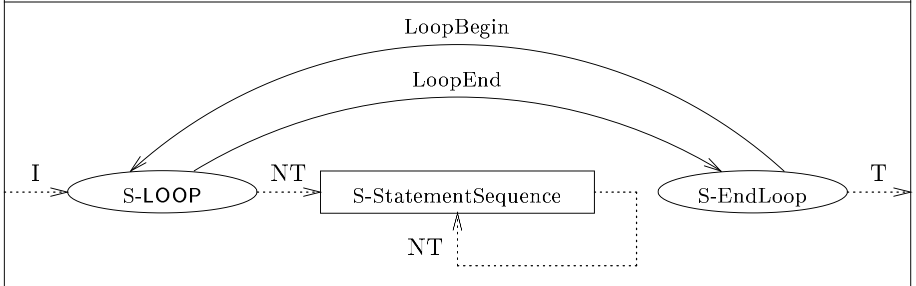
condition S-Expression.StaticType = BOOLEAN

REPEAT:

if Condition.VALUE = false **then**
  CurrentTask := FalseTask
else
  CurrentTask := NextTask
endif

Montage $\mathcal{O}_2$.M.9: Semantics of a repeat statement

LoopStatement ::= **LOOP StatementSequence EndLoop**
EndLoop = **END**
ExitStatement = **EXIT**



LOOP:

OldBlockEnd := GlobalBlockEnd
GlobalBlockEnd := LoopEnd

EndLoop:

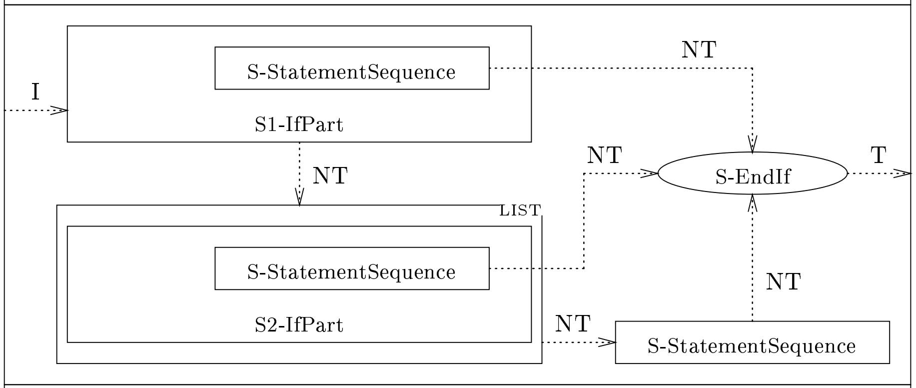
GlobalBlockEnd := LoopBegin.OldBlockEnd
CurrentTask := NextTask

EXIT:

CurrentTask := GlobalBlockEnd

Montage $\mathcal{O}_2$.M.10: Semantics of a loop statement

IfStatement ::= **IF ThenPart**
{ELSIF ThenPart}
[ELSE StatementSequence] EndIf
EndIf = **END**

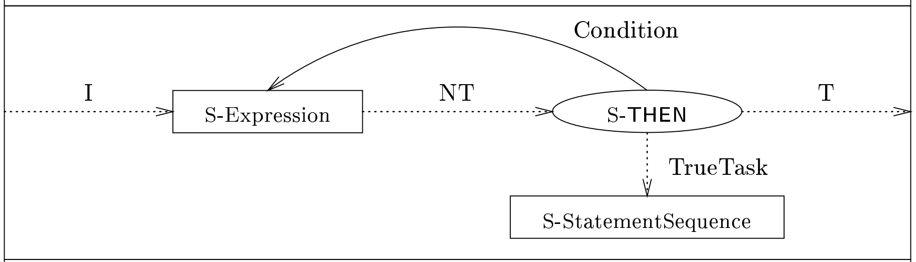


EndIf:

CurrentTask := NextTask

Montage $\mathcal{O}_2$.M.11: Semantics of an if statement

ThenPart ::= **Expression THEN StatementSequence**



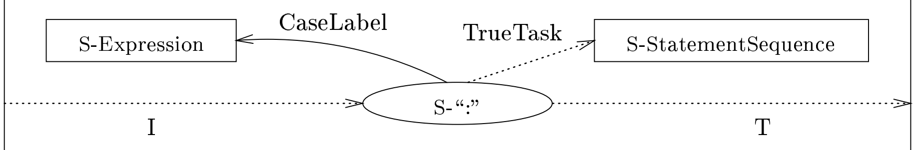
condition S-Expression.StaticType = BOOLEAN

THEN:

if Condition.VALUE = true **then**
CurrentTask := TrueTask
else
CurrentTask := NextTask
endif

Montage $\mathcal{O}_2$.M.12: Semantics of a then part

Case ::= **Expression ":" StatementSequence**



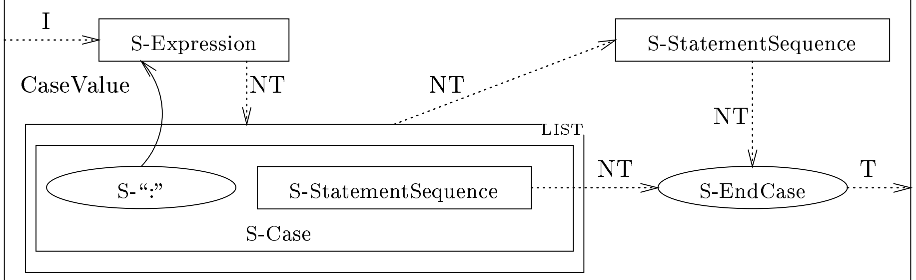
condition S-Expression.StaticType = SET **and** S-Expression.ConstValue $\neq$ undef

":"

if CaseValue.VALUE $\in$ CaseLabel.ConstValue **then**
   CurrentTask := TrueTask
else
   CurrentTask := NextTask
endif

Montage $\mathcal{O}_2$.M.13: Semantics of a case

CaseStatement ::= **CASE Expression OF**
Case {"|" Case}
[ELSE StatementSequence] EndCase
EndCase = **END**



if NoNode(S-StatementSequence) **then**
   DoAbort(S-StatementSequence) := true
   NoNode(S-StatementSequence) := false
endif

condition IntegerType(S-Expression.StaticType) **or** S-Expression.StaticType = CHAR

EndCase:

CurrentTask := NextTask

DoAbort:

CurrentTask := RunTimeError

Montage $\mathcal{O}_2$.M.14: Semantics of a case statement

3.7 Array Types

An array is a composed object that represents a sequence of elements that are all of the same type. The elements of an array are accessed using the Field function with the element–position as second argument. The numbering of the positions starts with zero. The type of an array fixes the length of the element sequence and defines the element type. The syntax for an array type with length n and element type t is

ARRAY n OF t

If t is again an array type, say of length m, the following shorthand notation is allowed to denote such a two dimensional array.

ARRAY n, m OF t

Syntactically this can be managed by introducing the name OfElementType for the part after the first dimension of an array type. The following productions

ArrayType	::=	ARRAY Expression OfElementType	($\mathcal{O}_2$.G.8)
OfElementType	=	NormalElementType | ShorthandArrayType	
NormalElementType	::=	OF Type	
ShorthandArrayType	::=	ShorthandArray Expression OfElementType	
ShorthandArray	=	“,”	

allow to treat the shorthand notation analogue to the normal one, by treating a ShorthandArray–task like an ARRAY–task. The Montages for ArrayType ($\mathcal{O}_2$.M.15) and for ShorthandArrayType ($\mathcal{O}_2$.M.16) work both as follows.

The static semantics guarantees that the length expression is constant and that its constant value is a positive integer. The LENGTH field is set to that constant value while the element type is set to the OfElementType part.

The create service of an array type allocates a composed object and a sequence of new objects of element type. The sequence of new elements are accessible as Field(0), Field(1), e.t.c.

A field Counter is used to iterate from 0 to (LENGTH -1). If the counter is undefined, it is initialized with 0 and a new element is created using the create service of the element type as follows:

CreateElement $\triangleq$ ($\mathcal{O}_2$.D.12)
  ElementType.Mode := create
  ElementType.Return := CurrentTask
  CurrentTask := ElementType

CreateElement sets the return address to the current task. Like this the control comes back and a next iteration step can be done. At each iteration step the newly created element is assigned to the field at the counter's position, the counter is incremented, and again a new element is created. If the iteration terminates, the last element is assigned to its field, the counter is reset to undef, and control is passed to the return task.

ArrayType ::= **ARRAY Expression OfElementType**

S-ARRAY.LENGTH := S-Expression.ConstValue

condition S-Expression.ConstValue $\geq$ 0 **and** IntegerType(S-Expression.StaticType)

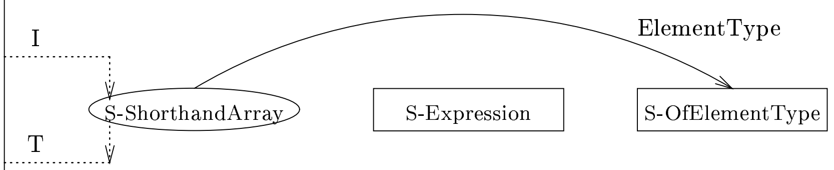
ARRAY, ShorthandArray:

```
if Mode = create then
   if Counter = undef then
      Counter := 0
      extend Composite with o
         Dest := o
      endextend
      CreateElement
   else
      (Dest.Field(Counter) := ElementType.Dest
      if Counter < (LENGTH - 1) then
         CreateElement
         Counter := Counter + 1
      else
         Counter := undef
         CurrentTask := Return
      endif
   endif
endif
```

```
if Mode = copy then
   if Counter = undef then
      Counter := 0
   else
      if Counter < LENGTH then
         CopyElement
         Counter := Counter + 1
      else
         Counter := undef
         CurrentTask := Return
      endif
   endif
endif
```

Montage $\mathcal{O}_2$.M.15: Semantics of an array type without shorthand notation.

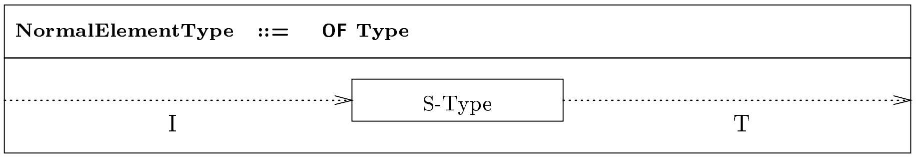
ShorthandArrayType ::= **ShorthandArray Expression OfElementType**
ShorthandArray = **“,”**



S-ShorthandArray.LENGTH := S-Expression.ConstValue

condition S-Expression.ConstValue $\geq$ 0 **and** IntegerType(S-Expression.StaticType)

Montage $\mathcal{O}_2$.M.16: Semantics of an array type with shorthand notation.



Montage $\mathcal{O}_2$.M.17: The help construction normal element type.

The copy service of an array type copies the values of the elements of its source to the values of the elements of its destination. The macro

CopyElement $\triangleq$ ($\mathcal{O}_2$.D.13)
  ElementType.Scr := Src.Field(Counter)
  ElementType.Dest := Dest.Field(Counter)
  ElementType.Return := CurrentTask
  CurrentTask := ElementType

calls the create service of the element type with the elements at position Counter. Unlike the standard copy macro $\mathcal{O}_1$.D.1 the return address is set to the current task. Like this, the iteration can be done as in the create case.

Arrays are pointable types, therefore we have to refine the synonym production PointableType ($\mathcal{O}_1$.G.5) and the universe PointableTypeTask ($\mathcal{O}_1$.C.2). After this refinements the pointer type Montages ($\mathcal{O}_1$.M.7, $\mathcal{O}_1$.M.8) are still valid.

PointableType = RecordType | ArrayType ($\mathcal{O}_2$.G.9)

PointableTypeTask = $\mathsf{RECORD} \cup \mathsf{ARRAY}$ ($\mathcal{O}_2$.C.7)

3.8 Indexed Designators

The elements of an array can be accessed using an indexed designator.

Indexed	::=	SequenceOfIndexed Expression IndexTask	($\mathcal{O}_2$.G.10)
IndexTask	=	"]"	
SequenceOfIndexed	=	DesignatorBracket | ShorthandIndexed	
DesignatorBracket	::=	Designator "["	
ShorthandIndexed	::=	SequenceOfIndexed Expression ShIndexTask	
ShIndexTask	=	","	

The textual part of the static analysis of both indexed and shorthand indexed designators is essentially the following macro, which does the implicit dereferencing for the typing.

$$\begin{array}{l} \mathrm{SetTypeOfIndexTask}(\mathit{selector}) \quad \triangleq \\ \quad \textbf{if } \mathrm{IsArray}(\mathrm{ArgTypeOfIndexed}) \textbf{ then} \\ \quad\quad \mathit{selector}.\mathrm{StaticType} := \mathrm{ArgTypeOfIndexed.ElementType} \\ \quad \textbf{elseif } \mathsf{POINTER}(\mathrm{ArgTypeOfIndexed}) \textbf{ then} \\ \quad\quad \mathit{selector}.\mathrm{StaticType} := \\ \quad\quad\quad \mathrm{ArgTypeOfIndexed.PointedType.ElementType} \\ \quad \textbf{endif} \end{array} \tag{$\mathcal{O}_2$.A.4}$$

where the type of the designator is

$$\mathrm{ArgTypeOfIndexed} \quad \triangleq \quad \mathrm{S\text{-}SequenceOfIndexed.StaticType} \tag{$\mathcal{O}_2$.A.5}$$

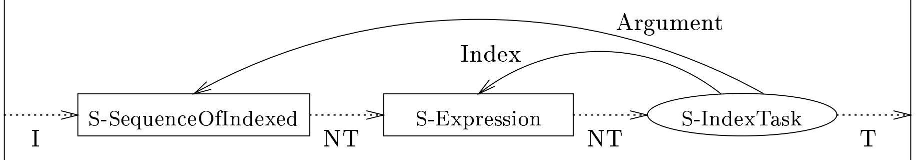
and the test for array types is

$$\mathrm{IsArray}(t) \quad \triangleq \quad \mathsf{ARRAY}(t) \textbf{ or } \mathrm{ShorthandArray}(t) \tag{$\mathcal{O}_2$.A.6}$$

The static semantics condition of the two Montages states essentially, that the argument designator is either an array or a pointer to an array.

The dynamic semantics of an index task (respectively of a shorthand index task) raises a run–time–error in the case of a range overflow and in the case of an attempt to dereference a Nil-pointer.

Indexed ::= **SequenceOfIndexed Expression IndexTask**
IndexTask = "]"
SequenceOfIndexed = **DesignatorBracket** | **ShorthandIndexed**



SetTypeOfIndexTask(S-IndexTask)

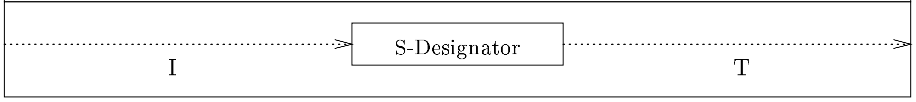
condition IntegerType(S-Expression.StaticType) **and**
(IsArray(ArgTypeOfIndexed) **or**
(POINTER(ArgTypeOfIndexed) **and**
IsArray(ArgTypeOfIndexed.ElementType)))

IndexTask, ShIndexTask:

```
if Index.VALUE ≥ StaticType.LENGTH or
   Index.VALUE < 0 then
      CurrentTask := RunTimeError
else
   if IsArray(Argument.StaticType) then
      SetTo(Argument.AssocOBJ.Field(Index.VALUE))
   elseif POINTER(Argument.StaticType) then
      if Argument.Value ≠ Nil then
         SetTo(Argument.VALUE.Field(Index.VALUE))
      else
         CurrentTask := RunTimeError
      endif
   endif
endif
```

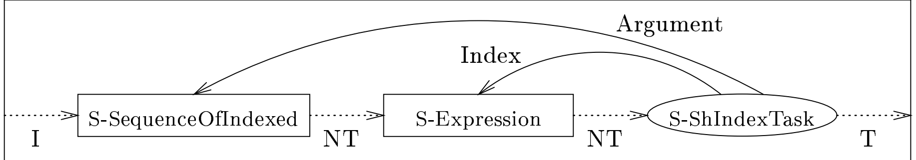
Montage $\mathcal{O}_2$.M.18: Semantics of an indexed designator.

DesignatorBracket ::= **Designator** "["



Montage $\mathcal{O}_2$.M.19: The help construction designator-bracket.

ShorthandIndexed ::= **SequenceOfIndexed Expression ShIndexTask**
ShIndexTask = ","



SetTypeOfIndexTask(S-ShIndexTask)

condition IntegerType(S-Expression.StaticType) **and**
(IsArray(ArgTypeOfIndexed) **or**
(POINTER(ArgTypeOfIndexed) **and**
IsArray(ArgTypeOfIndexed.ElementType)))

Montage $\mathcal{O}_2$.M.20: Semantics of a shorthand indexed designator.

4 Procedure Types, Declarations, and Calls

In its simplest form, a procedure is just a named sequence of statements, the so called procedure–body. A procedure call is a statement which executes the body. In addition Oberon procedures encompass

- the concept of a result: a procedure delivering a result can be used as expression;
- the concept of parameters;
- the concept of local types, variables and procedures.

In the following we specify procedures in two refinement levels. In the first (section 4.1) we abstract from parameters and explain all mechanisms needed for recursive calls, data encapsulation, and visibility scopes. In the second (section 4.2) we introduce actual and formal parameters, and how their typing is checked. Section 4.3 introduces open arrays. Section 4.4 shows how procedure types, variables, and the corresponding assignments work. Section 4.5 explains forward declarations and how the different procedure declarations are integrated in the declaration sequence. Finally 4.6 gives the specification of functions and the return statement.

4.1 A model without parameters

One of the aspects of procedures is the encapsulation of both code and data. The simple notion of procedure as named sequence of statements provides only encapsulation of code. The declaration of local variables, allows to encapsulate data in a private part of the store. Local types can be seen as encapsulation of the structure of local data, and local procedures allow to use the concept of encapsulation within a procedure.

Local declarations correspond to declaration sequences (see Montage $\mathcal{O}_1$.M.14 and refinements in $\mathcal{O}_2$.G.6 and $\mathcal{O}_4$.G.1). A code object (e.g. variable, type, procedure) which is locally declared in a procedure can only be referenced within that procedure.

The section of the code in which a code object can be referenced is called its scope. In our setting we model the different scopes as natural numbers. The scope of (code objects declared in) the program is 1. The scope of (code objects declared in) a procedure local to a program is 2. The scope of (code objects declared in) a procedure local to a procedure with scope n is $n + 1$. For each scope we introduce a new declaration table. All tables are represented by a binary dynamic function

$$\text{Table: Nat} \times \text{String} \rightarrow \text{CodeObject} \qquad (\mathcal{O}_3.\Sigma.1)$$

whose first argument is used to chose a scope. During the static analysis, we guarantee that a dynamic function

$$\text{Scope: Nat} \qquad (\mathcal{O}_3.\Sigma.2)$$

denotes the scope of the currently analyzed code. In all up to now given Montages, the declaration table is used on the current scope. Thus the old Montages are still valid if we redefine TABLE ($\mathcal{O}_1.\Sigma.6$) as follows:

$$\text{TABLE}(\textit{name}) \quad \triangleq \quad \text{Table(Scope, } \textit{name}) \qquad (\mathcal{O}_3.\text{A}.1)$$

If we enter during static analysis a new scope, the entries in the old declaration table are copied to the new one. Like this the global objects are visible until they are overwritten by a local declaration.

ProcObject = **CodeObject**

```
Scope := Scope + 1

TABLE(Name) := Self
Table(Scope + 1, Name) := Self
vary n over String
  satisfying n ≠ Name
    Table(Scope + 1, n) := TABLE(n)
endvary
```

condition **not** Predefined(Name)

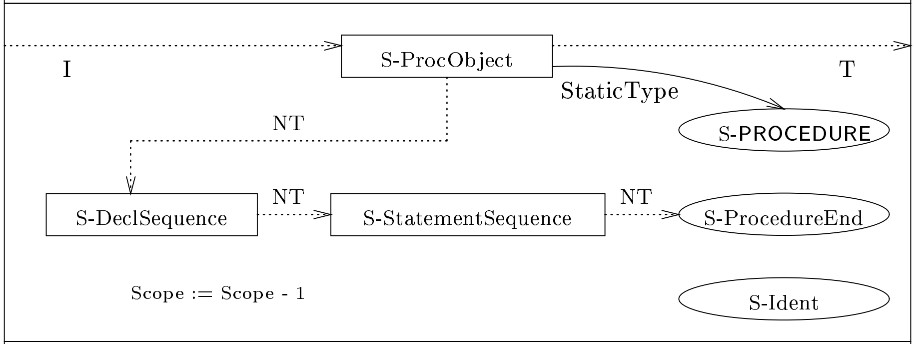
ProcObject:

```
CurrentTask := NextTask
```

Montage $\mathcal{O}_3$.M.1: The semantics of a procedure object.

ProcDeclaration ::= **PROCEDURE ProcObject** “;”
DeclSequence
[**BEGIN StatementSequence**] **ProcedureEnd Ident**

ProcedureEnd = **END**



condition S-ProcObject.Name = S-Ident.Name

ProcedureEnd:

RecLevel := RecLevel -1
CurrentTask := CALLER.NextTask

Montage $\mathcal{O}_3$.M.2: The semantics of a simplified procedure declaration.

The static analysis of a procedure declaration (Montage $\mathcal{O}_3$.M.2) analyses first its procedure object. The static analysis of this procedure object (Montage $\mathcal{O}_3$.M.1) increments the scope, and copies the declaration tables of the old scope. In addition it enters itself in the procedure declaration table of both the old and the new scope. The static analysis of the procedure declaration continues by analyzing the local declaration sequence and the statement sequence. After this the increased scope is decremented to its old value (see textual part of static analysis of Montage $\mathcal{O}_3$.M.2). The token S-PROCEDURE serves as representant of the type of the procedure.

Before we explain the dynamic semantics of a procedure call we shortly describe the general problems that have to be considered for the specification of recursive procedure calls. During execution four kinds of results have been produced by the already executed tasks and can be reused by the tasks that are still going to be executed:

– The result of the evaluation of an expression task is assigned to its VALUE field.
– The associated object of a variable (or parameter) object is stored in its AssocOBJ field.
– The calculated associated object (and eventually value) of a designator task is stored in its AssocOBJ (and eventually VALUE) field.
– The actual length of an array type is stored in its LENGTH field.

If the same task is executed twice (e.g. if it is part of a loop) the results stored by VALUE and AssocOBJ are overwritten. While ($\mathcal{O}_2$.M.8), repeat ($\mathcal{O}_2$.M.9), and loop statements ($\mathcal{O}_2$.M.10) rely on this behavior.

As mentioned procedures are named units of code which can be executed by calling them. Such a call is an ordinary statement. As a procedure contains statements, it can contain a call to itself as well. This leads to the situation that the tasks of the itself calling procedure are executed twice. But the results of the first execution should here not be overwritten by the recursive call: they are still needed after the recursive call.

To resolve the above problem we use the same technique as for the scopes. The recursion levels are represented by natural numbers and a dynamic unary function

$$\text{RecLevel: Nat} \qquad (\mathcal{O}_3.\Sigma.3)$$

is initialized with 0 and is guaranteed to hold always the recursion level of the code in execution, e.g. the current recursion level. For each recursion level there are three unary functions for the associated object, the value, and the length. Again these functions are represented by binary functions whose first argument is used to chose the recursion level.

$$\text{AssoccObj: Nat} \times (\text{VarObject} \cup \text{DesignatorTask}) \rightarrow \text{Object} \qquad (\mathcal{O}_3.\Sigma.4)$$

$$\text{RecLevelValue: Nat} \times \text{ExpressionTask} \rightarrow \text{OberonValue} \qquad (\mathcal{O}_3.\Sigma.5)$$

$$\text{RecLevelLength: Nat} \times \mathsf{ARRAY} \rightarrow \text{Nat} \qquad (\mathcal{O}_3.\Sigma.6)$$

Since all presented Montages access the associated objects, values, and length only on the current recursion level, we can refine the old functions to macros as follows:

$$n.\text{AssocOBJ} \quad \triangleq \quad \text{AssoccObj (RecLevel, } n) \qquad (\mathcal{O}_3.\text{D}.1)$$

$$n.\text{VALUE} \quad \triangleq \quad \text{RecLevelValue(RecLevel, } n) \qquad (\mathcal{O}_3.\text{D}.2)$$

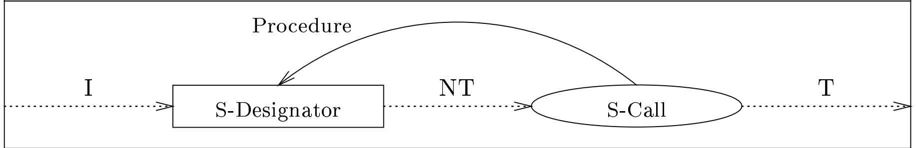
$$n.\text{LENGTH} \quad \triangleq \quad \text{RecLevelLength(RecLevel, } n) \qquad (\mathcal{O}_3.\text{D}.3)$$

In order to understand how a procedure call works, we have to know how a designator works if it references a procedure. As for normal designators the static analysis of a simple designator ($\mathcal{O}_1$.M.2) copies the entry in the table to its declaration field and sets the static type to the static type of the declaration. For procedure designators, the dynamic semantics of a simple sets the VALUE directly to the declared procedure.

ADD ELSIF-CLAUSE TO DYNAMIC SEMANTICS ($\mathcal{O}_3$.D.4)
OF SIMPLE $\mathcal{O}_1$.M.2:
    elseif ProcObject(Decl) **then**
        VALUE := Decl

All other designators treat references to procedures correctly and must not be refined.

ProcedureCall ::= **Call Designator**
Call =



condition PROCEDURE(S-Designator.StaticType)

Call:

RecLevel := NewRecLevel
vary o **over** VarObject ∪ ParamObject ∪ DesignatorTask
  AssoccObj (NewRecLevel, o) := o.AssocOBJ
endvary
vary o **over** ExpressionTask
  RecLevelValue(NewRecLevel, o) := o.VALUE
endvary
vary o **over** ARRAY
  RecLevelLength(NewRecLevel, o) := o.LENGTH
endvary
Caller(NewRecLevel) := CurrentTask
CurrentTask := Procedure.VALUE
where NewRecLevel $\triangleq$RecLevel + 1

Montage $\mathcal{O}_3$.M.3: Semantics of a simplified procedure call.

A procedure call (Montage $\mathcal{O}_3$.M.3) consists of a designator, which must be of procedure type. The dynamic semantics rule increments the recursion level with one and the values of AssocOBJ, VALUE, and LENGTH are copied to the new recursion level. The unary dynamic function

$$\text{Caller: Nat} \rightarrow \text{Call} \qquad (\mathcal{O}_3.\Sigma.7)$$

is used to store the Call-token that invocated a recursion level , and the macro

$$\text{CALLER} \quad \triangleq \quad \text{Caller(RecLevel)} \qquad (\mathcal{O}_3.\text{D}.5)$$

denotes the caller that invocated the current recursion level. The dynamic semantics of a Call-token sets the caller of the new recursion level to itself and passes control to the procedure.

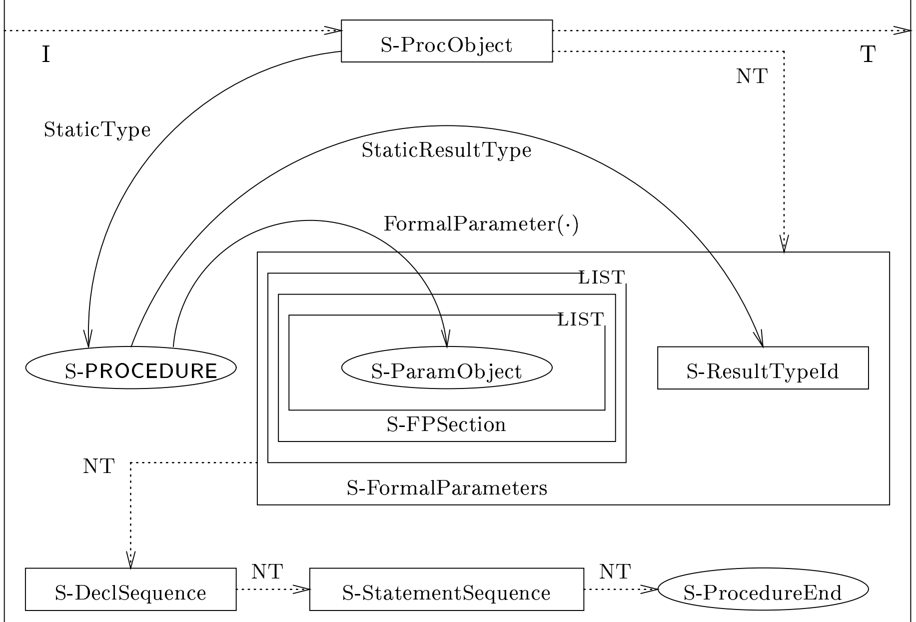
After execution of the procedure body the control is given back to the caller and the recursion level decremented with one. This is, for instance, done by the dynamic semantics of a ProcedureEnd-task, see Montage $\mathcal{O}_3$.M.2.

4.2 Introducing parameters

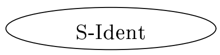
In this section we add formal parameters to the procedure declaration and actual parameters to the procedure call. The basic mechanisms remain the same and most of the specification is concerned with the typing constraints of the parameters.

ProcDeclaration ::= **PROCEDURE ProcObject** [**FormalParameters**] **";"**
DeclSequence
[**BEGIN StatementSequence**] **ProcedureEnd Ident**

ProcedureEnd = **END**



Scope := Scope - 1



if not NoNode(S-FormalParameter.S-"(") **then**
S-PROCEDURE.HasBrackets := true
else
S-PROCEDURE.HasBrackets := false
endif

condition S-ProcObject.Name = S-Ident.Name
and NamesInFormalParamsAndDeclSequenceDifferent

ProcedureEnd:

RecLevel := RecLevel -1
CurrentTask := CALLER.NextTask

Montage $\mathcal{O}_4$.M.1: The semantics of a procedure declaration with formal parameters.

The refined specification of a procedure declaration (Montage $\mathcal{O}_4$.M.1) encompasses formal parameters. The static analysis defines graphically the binary function

$$\text{FormalParameter: PROCEDURE} \times \text{Nat} \rightarrow \text{ParamObject} \qquad (\mathcal{O}_4.\Sigma.1)$$

which maps the PROCEDURE–token to the formal parameters and the field

$$\text{StaticResultType: PROCEDURE} \rightarrow \text{TypeTask} \qquad (\mathcal{O}_4.\Sigma.2)$$

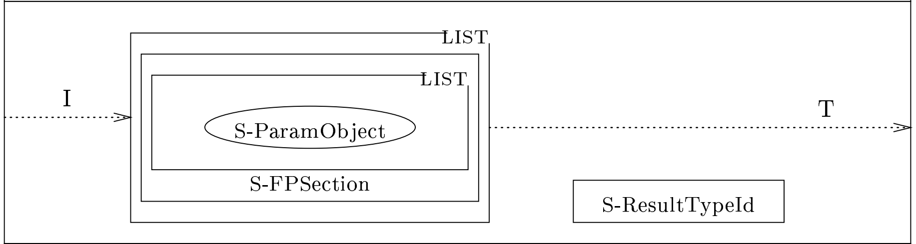
which maps the PROCEDURE–token to the result type. A PROCEDURE–token together with these two functions forms a procedure type. A procedure type may as well be given explicitly. This is explained in detail by the procedure–type Montage $\mathcal{O}_4$.M.6. The type ResultTypeId is guaranteed to be unstructured (see $\mathcal{O}_4$.A.3).

The static semantics of a procedure declaration guarantees that the names of the formal parameters are different from the names of the code objects in the declaration sequence of the procedure.

NamesInFormalParamsAndDeclSequenceUnique $\triangleq$ ($\mathcal{O}_4$.C.1)
for all p: ParamObject
satisfying p ParamObjectOf S-FormalParameters **holds**
for all c: CodeObject
satisfying c CodeObjectOf S-DeclSequence **holds**
p.Name $\neq$ c.Name

where the definition of ParamObjectOf will be given after the formal parameters Montage and a new definition of CodeObjectOf, accounting as well for procedure objects, will be given later in this section ($\mathcal{O}_4$.C.15).

FormalParameters ::= "(" [**FPSection**{";" **FPSection**}] ")" [":" **ResultTypeId**]



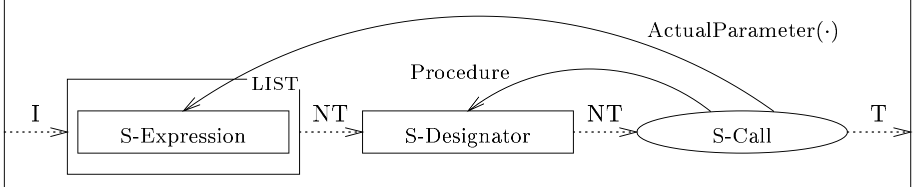
condition **for all** p1: ParamObject
satisfying p1 ParamObjectOf Self **holds**
for all p2: ParamObject
satisfying p2 ParamObjectOf Self **holds**
p1 $\neq$ p2 **implies** p1.Name $\neq$ p2.Name
and (**not** NoNode(S-ResultTypeId)
implies UnstructuredType(S-ResultTypeId))

Montage $\mathcal{O}_4$.M.2: The semantics of formal parameters.

The static semantics of formal parameters (Montage $\mathcal{O}_4$.M.2) guarantees that all parameter objects have different names. We use again the macro ParamObjectOf which takes as second argument a FormalParameters-node and is defined as follows.

p ParamObjectOf *fp* $\triangleq$ ($\mathcal{O}_4$.C.2)
exists FPs **in list** *fp*.S-FPSection
such that *p* **in list** FPs.S-ParamObject

ProcedureCall ::= **Call Designator**["("[**Expression** {"," **Expression**]")"]
Call =



condition PROCEDURE(S-Designator.StaticType)
and CompatibleParameters

Call:

```
vary o over VarObject ∪ ParamObject ∪ DesignatorTask
  AssoccObj (NewRecLevel, o) := o.AssocOBJ
endvary
vary o over ExpressionTask
  RecLevelValue(NewRecLevel, o) := o.VALUE
endvary
vary o over ARRAY
  RecLevelLength(NewRecLevel, o) := o.LENGTH
endvary
Caller(NewRecLevel) := CurrentTask
RecLevel := NewRecLevel
CurrentTask := Procedure.VALUE
where NewRecLevel ≜RecLevel + 1
```

Montage $\mathcal{O}_4$.M.3: Semantics of a procedure call with actual parameters.

A call of procedures with parameters (Montage $\mathcal{O}_4$.M.3) consists of a designator and a list of expressions, the so called actual parameters. The static semantics of a procedure call guarantees that actual and formal parameters either both have brackets or not, and that the actual parameters are correctly typed with respect to the formal parameters of the procedure type. The actual parameters are accessible as the list of expressions and the formal parameters are accessible via the enumeration function FormalParameter. The following macro CompatibleParameters checks whether the two lists have the same length, and whether all elements are CallCompatible.

CompatibleParameters $\triangleq$ ($\mathcal{O}_4$.C.3)
  S-Expression.ListLength =
    S-Designator.StaticType.FormalParameterLength
  and
  NoNode(S-"(") = S-Designator.StaticType.HasBracket
  and
  for all e **in list** S-Expression **holds**
    CallCompatible(e,
      S-Designator.StaticType.FormalParameter(e.Position))

The macro CallCompatible is defined differently for variable parameter objects, which are members of the universe VariableParamObject, and for value parameter objects, which are members of the universe ValueParamObject.

$$\text{ParamObject} = \text{VariableParamObject} \cup \text{ValueParamObject} \qquad (\mathcal{O}_4.\Sigma.3)$$

In short, a variable parameter is a reference to a variable, which can be updated in the procedure. Thus the static semantics of variable parameters must guarantee, that the actual parameter is a designator with an associated object, and not only an expression. In addition the types of this objects must be VariableParamCompatibleTypes. For this refinement level VariableParamCompatibleTypes is defined as the identity:

$$\text{VariableParamCompatibleTypes}\ (t1,\ t2) \quad \triangleq \quad t1\ =\ t2 \qquad (\mathcal{O}_4.\text{C}.4)$$

A value parameter is an independent variable, which is initialized with the value of the actual parameter. Thus actual parameters of value parameters may be arbitrary expressions whose type is assignment compatible with the formal parameter.

CallCompatible(e, p) $\triangleq$ ($\mathcal{O}_4$.C.5)
  VariableParamObject(p) **implies**
    VariableParamCompatibleTypes(e.StaticType, p.StaticType)
    and DesignatorTask(e)
    and (Simple(e) **implies** VarObject(e.Decl))
  and
  ValueParamObject(p) **implies**
    e.StaticType AssignableTo p.StaticType

The formal semantics of variable and value parameters is given by Montage $\mathcal{O}_4$.M.4.

The static analysis sets the static type of all parameter objects and puts them either in the universe VariableParamObject or in the universe ValueParamObject. The dynamic semantics of both types of parameters uses the macro

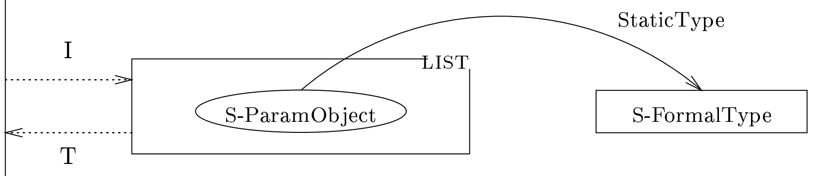
MyActualParameter $\triangleq$ ($\mathcal{O}_4$.D.1)
  CALLER.ActualParameter(FormalParameterPosition)

to refer to their corresponding actual parameter. As noted a variable parameter just copies the associated object reference of its actual parameter, whereas the value parameter allocates a new object of its type and copies the value(s) of its actual parameter to this new object. The macro

SetLENGTH $\triangleq$ ($\mathcal{O}_4$.D.2)
  if ARRAY(StaticType) **then**
    StaticType.LENGTH :=
      MyActualParameter.StaticType.LENGTH
  endif

sets the LENGTH of an array typed parameter to the LENGTH of its actual parameter. This is needed in order to deal with open array types.

FPSection ::= [**VAR**] **ParamObject{"," ParamObject} ":" FormalType**
FormalType = **OpenArrayType** | **TypeIdent**




```
if NoNode(S-VAR) then
   vary p over list S-ParamObject
      ValueParamObj(p) := true
   endvary
else
   vary p over list S-ParamObject
      VariableParamObj(p) := true
   endvary
endif
```

VariableParamObj:

```
SetLENGTH
AssocOBJ:= MyActualParameter.AssocOBJ
CurrentTask := NextTask
```

ValueParamObj:

```
SetLENGTH
StaticType.CREATE() ;
   AssocOBJ:= NewObject
   if UnstructuredType(StaticType) then
      NewObject.Value := MyActualParameter.VALUE
      CurrentTask := NextTask
   else
      StaticType.COPY(MyActualParameter.AssocOBJ, NewObject)
   endif
```

Montage $\mathcal{O}_4$.M.4: Semantics of Value and Variable Parameters.

4.3 Open Arrays

Open arrays (Montage $\mathcal{O}_4$.M.5) can be used as the formal type of parameters, and as element types of open arrays. The model must be refined at several places in order to treat open arrays correctly:

– Assignments between open arrays are not allowed. Thus we have to redefine the definition of AssignableTo ($\mathcal{O}_2$.C.1) by conjuncting the old definition with

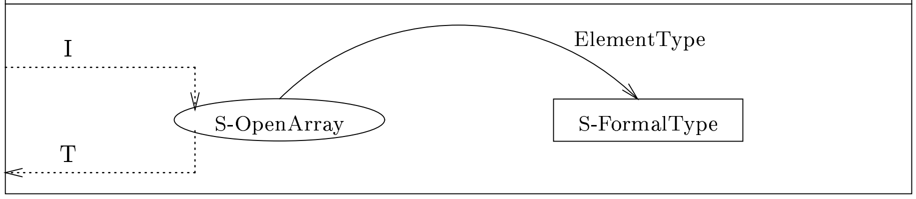
  CONJUNCT ASSIGNABLETO $\mathcal{O}_2$.C.1 WITH: ($\mathcal{O}_4$.C.6)
  not OpenArray(*t1*) **and not** OpenArray(*t2*)

– In contrast, for a formal parameter of open array type any array with the same dimensionality and the same element type may be given as actual parameter. In order to allow for this we have to weaken the definition of CallCompatible macro ($\mathcal{O}_4$.C.5) by disjuncting it with

  DISJUNCT CALLCOMPATIBLE $\mathcal{O}_4$.C.5 WITH: ($\mathcal{O}_4$.C.7)
  OpenArrayParamCompatibleTypes
  (*e*.StaticType, *p*.StaticType)

OpenArrayParamCompatibleTypes(*t1*, *t2*) $\triangleq$ ($\mathcal{O}_4$.C.8)
  IsArray(*t1*) **and** OpenArray(*t2*) **and**
   (*t1*.ElementType = *t2*.ElementType
   or OpenArrayParamCompatibleTypes
     (*t1*.ElementType, *t2*.ElementType)

OpenArrayType ::= OpenArray OF FormalType
OpenArray = ARRAY



Montage $\mathcal{O}_4$.M.5: Semantics of an open array type.

4.4 Procedure Types and Variables

In Oberon one can declare variables of procedure types. As noted above a procedure type consists of a PROCEDURE-token and the two functions FormalParameter and StaticResultType. The procedure type Montage ($\mathcal{O}_4$.M.6) corresponds exactly to that part of the procedure declaration Montage ($\mathcal{O}_4$.M.1). The dynamic semantics corresponds to ground types.

Designators of procedure type may be used in comparisons, assignments, and as parameters, leading to the following refinements:

- Designators of procedure type may be compared for equality and inequality, if their procedure types are identical (see table of S-o2ACalculatedType). Identity of procedure types is defined as follows:

  IdenticProcTypes(*t1*, *t2*) $\triangleq$ ($\mathcal{O}_4$.C.9)
    t1 $\neq$ undef **and**
    t1.StaticResultType = *t2*.StaticResultType **and**
    t1.FormalParameterLength = *t2*.FormalParameterLength
    and
    for all i: Nat
     satisfying $1 \leq$ i $\leq$ *t1*.FormalParameterLength **holds**
      IdenticFormalTypes
       (*t1*.FormalParameter(i).StaticType,
       t2.FormalParameter(i).StaticType)

  IdenticFormalTypes(*t1*, *t2*) $\triangleq$ ($\mathcal{O}_4$.C.10)
    t1 = *t2* **or**
     OpenArray(*t1*) **and** OpenArray(*t2*) **and**
     IdenticFormalTypes(*t1*.ElementType, *t2*.ElementType)

– A designator of procedure type may be assigned to another one if the types are identical.

DISJUNCT AssignableTo $\mathcal{O}_4$.C.6 WITH: ($\mathcal{O}_4$.C.11)
IdenticProcTypes(*t1*, *t2*)

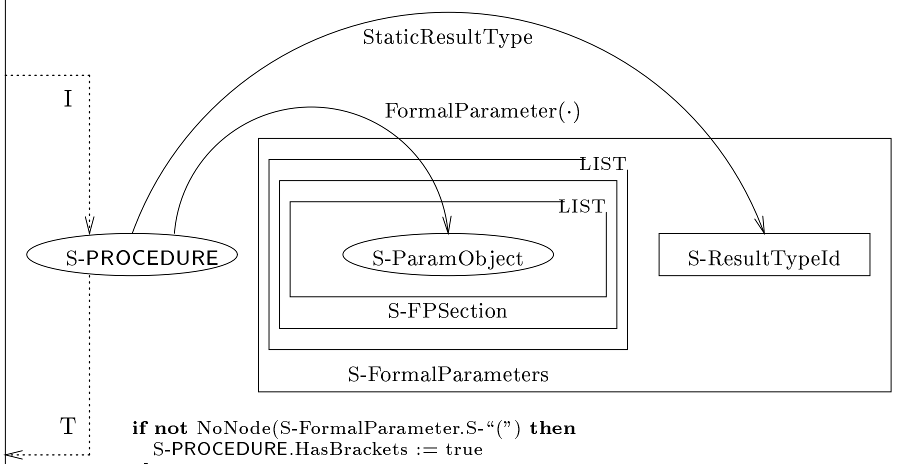
In addition, if the right-hand-side of the assignment is a procedure constant, this constant must not be a predefined procedure. Thus we need to add to the static semantics of the assignment (Montage $\mathcal{O}_1$.M.3 and refinements given after $\mathcal{O}_2$.G.2) the following.

CONJUNCT CONDITION OF Assignment $\mathcal{O}_1$.M.3 WITH: ($\mathcal{O}_4$.C.12)
ProcOrForwardObject(S-Expression.ConstValue) **implies**
not Predefined(S-Expression.ConstValue.Name)

– If a parameter is of procedure type the same rules apply as for other types. Nonetheless we have to redefine the definition of the macro VariableParam-TypeCompatible ($\mathcal{O}_4$.C.4), since identity of procedure types is expressed by the macro IdenticProcTypes.

VariableParamCompatibleTypes (*t1*, *t2*) $\triangleq$ ($\mathcal{O}_4$.C.13)
t1 = *t2* **or** IdenticProcTypes(*t1*, *t2*)

ProcedureType ::= **PROCEDURE** [**FormalParameters**] **“;”**




```
if not NoNode(S-FormalParameter.S-“(”) then
   S-PROCEDURE.HasBrackets := true
else
   S-PROCEDURE.HasBrackets := false
endif
```

ProcedureType:

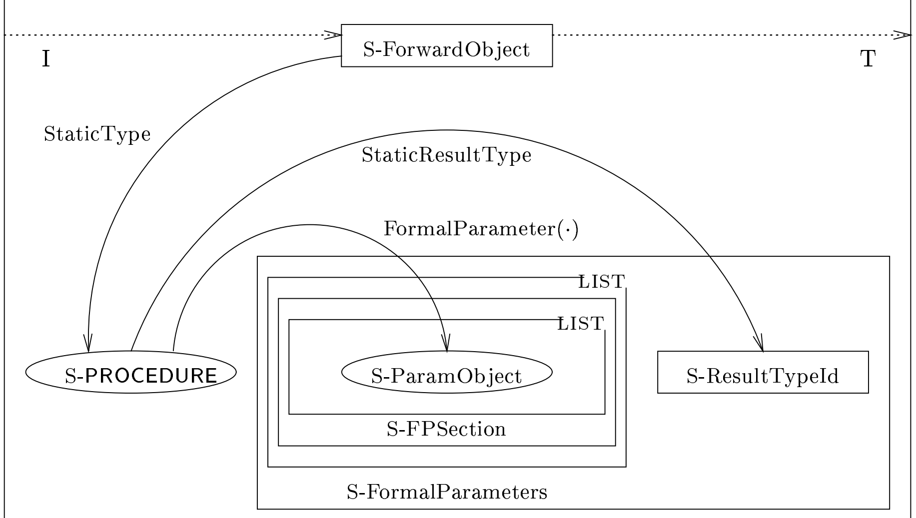
```
if Mode = copy then
   Dest.VALUE := Scr.VALUE
   CurrentTask := Return
endif
```

```
if Mode = create then
   extend Primitive with o
      Dest := o
      CurrentTask := Return
   endextend
endif
```

Montage $\mathcal{O}_4$.M.6: Semantics of a procedure type.

4.5 Forward Declarations

ForwardDeclaration ::= **PROCEDURE "↑" ForwardObject** [**FormalParameters**]



if not NoNode(S-FormalParameter.S-"(") **then**
  S-PROCEDURE.HasBrackets := true
else
  S-PROCEDURE.HasBrackets := false
endif

Montage $\mathcal{O}_4$.M.7: Semantics of a forward declaration.

ForwardObject = **CodeObject**

TABLE(Name) := Self

condition **not** Predefined(Name)

ForwardObject:
  CurrentTask := NextTask

Montage $\mathcal{O}_4$.M.8: Semantics of a forward object.

In order to allow for fast one–pass compilers Oberon restricts references to code objects which have been declared earlier. But on the other hand it should be possible that two procedures call them in a mutual recursive way, e.g. procedure A calls procedure B which in turn calls A. In such a situation it is required to give a forward declaration (Montage $\mathcal{O}_4$.M.7) of the procedure. A forward declaration contains the procedure type, and a the name of the procedure. The identifier is a forward object, which enters itself in the TABLE (see $\mathcal{O}_3$.M.1).

Both procedure and forward declarations are part of the declaration sequence ($\mathcal{O}_1$.M.14, and refinement in $\mathcal{O}_2$.G.6). Since procedure use types, variables, and constants, their declarations are put at the end of the declaration sequence. We postfix the grammar of the declaration sequence with

POSTFIX R-H-S OF DECLSEQUENCE ($\mathcal{O}_4$.G.1)
PRODUCTION $\mathcal{O}_2$.G.6 WITH:
{ProcOrForwardDecl ";" }

ProcOrForwardDecl ::= ProcDeclaration | ($\mathcal{O}_4$.G.2)
ForwardDeclaration

The dynamic semantics of a forward declaration just passes the control to the first task of the real procedure declaration. The static analysis of the declaration sequence sets the next task of a forward declaration to its procedure declaration.

BLOCK STATIC ANALYSIS OF DECLSEQUENCE $\mathcal{O}_1$.M.14 WITH: ($\mathcal{O}_4$.A.1)
vary fd **over list** S-ProcOrForwardDecl
  satisfying ForwardDeclaration(fd)
    fd.S-ForwardObject.NextTask :=
      TABLE(fd.S-ForwardObject.Name)
endvary

The static semantics of a declaration sequence guarantees that there is for each forward declaration a procedure declaration with the same name and identical type.

CONJUNCT CONDITION OF DECLSEQUENCE $\mathcal{O}_1$.M.14 WITH: ($\mathcal{O}_4$.C.14)
for all fd **in list** S-ProcOrForwardDecl
  satisfying ForwardDeclaration(fd) **holds**
    exists pd **in list** S-ProcOrForwardDecl
      satisfying ProcDeclaration(pd) **such that**
        fd.S-ForwardObject.Name =
          pd.S-ProcObject.Name **and**
        IdenticProcTypes
          (fd.S-PROCEDURE, pd.S-PROCEDURE)

The macro CodeObjectOf ($\mathcal{O}_1$.C.5, refinement in $\mathcal{O}_2$.C.4) must be refined in order to yield true for procedure objects as well.

DISJUNCT DEFINITION OF CODEOBJECTOF $\mathcal{O}_2$.C.4 WITH: ($\mathcal{O}_4$.C.15)
exists pd **in list** ds.S-ProcOrForwardDecl
  satisfying ProcDeclaration(pd) **such that**
    i = pd.S-ProcObject

What remains to do is a small, but subtle point. Designator of procedure types can be compared , and they should be considered as equal, if their values point either to the same object, or to a forward object and to the corresponding procedure object. Thus we have to redefine the function Apply ($\mathcal{O}_2.\Sigma.7$) for "="-operators as follows:

for $op \in$ "=": $(\mathcal{O}_4.\mathrm{D}.3)$
Apply(*op*, *left*, *right*) $\triangleq$
left = *right* **or**
(ForwardObject(*left*) **and** *left*.NextTask = *right* **or**
ForwardObject(*right*) **and** *right*.NextTask = *left*)

and analogously for "#"-operators.

4.6 Functions and the Return Statement

A procedure call can be used as statement, if it has no result type or as expression if it has a result type. If it is used as expression, it is a synonym of Factor. The test whether a call is correctly used as statement or expression looks as follows:

CONJUNCT CONDITION OF PROCEDURECALL $\mathcal{O}_4$.M.3 WITH: $(\mathcal{O}_4.\mathrm{C}.16)$
(Statement(Self) **implies** NoNode(CallResultType)
and
(Factor(Self) **implies** TypeTask(CallResultType)

where

CallResultType $\triangleq$ $(\mathcal{O}_4.\mathrm{C}.17)$
S-Designator.StaticType.StaticResultType

The static analysis of the procedure call must in addition set the static type:

BLOCK STATIC ANALYSIS OF PROCEDURECALL $\mathcal{O}_4$.M.3 WITH: $(\mathcal{O}_4.\mathrm{A}.2)$
if Factor(Self) **then**
S-Call.StaticType := CallResultType
endif

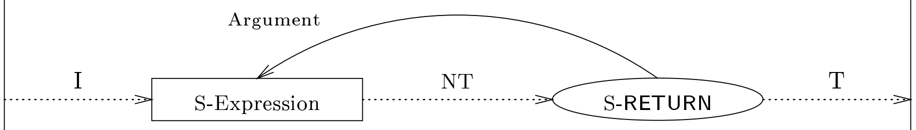
The return statement (Montage $\mathcal{O}_4$.M.9) is used within the body of a procedure declaration. Its static semantics checks whether it is part of a procedure declaration, e.g. whether the scope is larger then 0, and whether the type of its argument is assignment compatible to the result type. The result type of a scope is denoted by

ResultType: Nat $\rightarrow$ TypeTask $(\mathcal{O}_4.\Sigma.4)$
ResultTYPE $\triangleq$ ResultType(Scope)

This function is set by the ResultTypeId, whose Montage is obtained from the type identifier Montage ($\mathcal{O}_1$.M.6) by adding to the static analysis an update.

RESULTTYPEID MONTAGE IS $(\mathcal{O}_4.\mathrm{A}.3)$
TYPEIDENT MONTAGE $\mathcal{O}_1$.M.6 WHERE
BLOCK STATIC ANALYSIS WITH:
ResultTYPE := TABLE(Name)

ReturnStatement ::= **RETURN** [**Expression**]



ReturnStatementExists := true

condition Scope > 0 **and**
(ResultTYPE ≠ undef **implies**
S-Expression.StaticType AssignableTo ResultTYPE) **and**
(ResultTYPE = undef **implies** NoNode(S-Expression))

RETURN:

if not NoNode(Argument) **then**
CALLER.RefLevelValue(RecLevel -1) := Argument.VALUE
endif
RecLevel := RecLevel - 1
CurrentTask := CALLER.NextTask

Montage $\mathcal{O}_4$.M.9: Semantics of a return statement.

The static analysis of the return statement sets a boolean function

ReturnStatementExists: Boolean ($\mathcal{O}_4.\Sigma.5$)

which is needed to test the static semantics condition that there must be at least one return statement in the body of a function declaration.

CONJUNCT CONDITION OF PROCDECLARATION $\mathcal{O}_4$.M.1 WITH: ($\mathcal{O}_4$.C.18)
not NoNode(S-FormalParameters.S-ResultTypeId) **implies**
ReturnStatementExists

The static analysis of a procedure declaration must reset both the ResultTYPE and the ReturnStatementExists.

BLOCK STATIC ANALYSIS OF PROCDECLARATION $\mathcal{O}_4$.M.1 WITH: ($\mathcal{O}_4$.A.4)
ResultTYPE := undef
ReturnStatementExists := false

Finally the dynamic semantics of a return task (Montage $\mathcal{O}_4$.M.9) sets the value of its caller, decrements the recursion level and passes control to the next task of the caller.

5 Modules and Data Type Extension

In this section we specify Oberon support for modular programming and data type extensions.

5.1 Modules

ModuleList ::= Module { Module } $(\mathcal{O}_5.G.1)$

A list of modules replaces the program construct $(\mathcal{O}_1.M.15)$ as start symbol of our language. Like a program a module contains a declaration sequence and a statement sequence, called the body of the module. In addition a module contains a list of imports of other modules. Within a module exported code objects of imported modules can be use. A code object is exported if it is marked with an asterix "*".

CodeObject = ExportIdent $(\mathcal{O}_5.G.2)$

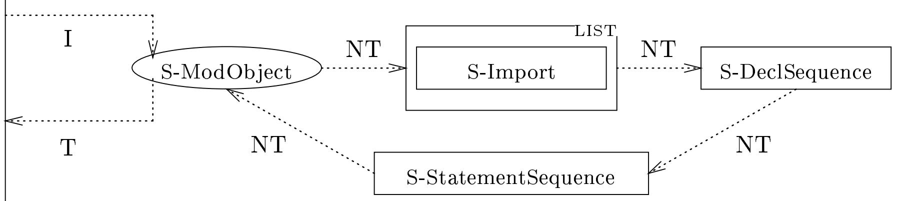
For simplicity we assume that the "*"-mark of an export identifier is recognized by the scanner, and that the name and the export mark are available as independent micro-syntax functions Name and Exported.

Name: ExportIdent → String $(\mathcal{O}_5.\Sigma.1)$
Exported: ExportIdent → Boolean

Like this the name of an export identifier is accessed like the name of a normal identifiers, and the existing specifications can remain valid.

The static analysis of a module (Montage $\mathcal{O}_5$.M.1) updates the field function such, that the module object corresponds to a record, with all exported code objects as fields[2]. Thus an exported object of a module can be accessed by applying the field function to its module object.

Module = **MODULE ModObject ";"**
[IMPORT [Import {"," Import}]] DeclSequence
[BEGIN StatementSequence] END Ident "."




```
vary c over CodeObject
  satisfying c CodeObjectOf S-DeclSequence and c.Exported
    if TypeObject(c) then
      S-ModObject.Field(c.Name) := TABLE(c.Name)
    else
      S-ModObject.Field(c.Name) := c
    endif
endvary
```

condition S-ModObject.Name = S-Ident.Name

Montage $\mathcal{O}_5$.M.1: Semantics of a module.

[2] As usual an exception are types that are entered in the table instead of their type objects (see as well Montage $\mathcal{O}_1$.M.5).

ModObject = **Ident**	

	ModuleTable(Name) := Self
	Scope := 0
	TABLE(Name) := Self
	vary n **over** String
	  satisfying not Predefined(n)
	    Table(0, n) := undef
	endvary
	CurModule := Self
	StaticType := Self
	ParentModule := Self

condition	ModuleTable(Name) = undef **and not** Predefined(Name)

ModObject:	**if** Initialised **then**
	  CurrentTask := Return
	else
	  Initialised := true
	  CurrentTask := NextTask
	endif

Montage $\mathcal{O}_5$.M.2: Semantics of a module object.

A module object enters itself in the table

$$\text{ModuleTable: String} \rightarrow \text{ModObject} \qquad (\mathcal{O}_5.\Sigma.2)$$

sets the scope to 0, and purges the non-predefined entries in the table. The function

$$\text{CurModule: ModObject} \qquad (\mathcal{O}_5.\Sigma.3)$$

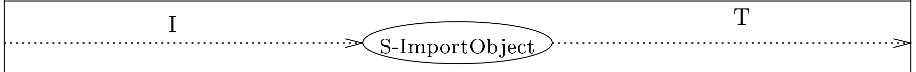
is set to the currently analyzed module. In the module Montage $\mathcal{O}_5$.M.1 we see that the module object is the anchor of a NextTask-linked ring list. The dynamic semantics of the module object sets the boolean field

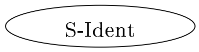
$$\text{Initialised: ModObject} \rightarrow \text{Boolean} \qquad (\mathcal{O}_5.\Sigma.4)$$

and passes control to the next task. If Initialised is already set, the module object passes control directly back to its return task. Like this it is guaranteed that each module is initialized only once.

The dynamic semantics of an import (Montage $\mathcal{O}_5$.M.3) passes control to the imported module and sets the return task of that module to its next task. Syntactically an import consists of an import object and an optional identifier. If the identifier is present, it denotes the name of the imported object, and the import object denotes a second name, the so called *local name*, which will be used within the module to refer to the imported object. If the optional identifier is not present, the import object denotes both the global and the local name of the imported object. The static analysis of an import enters the imported object under its local name in the TABLE. The static semantics guarantees that the imported module is in the ModuleTable.

```
Import          ::=  ImportObject [":" Ident]
ImportObject    =    Ident
```





if NoNode(S-Ident) **then**
  Decl := ModuleTable(S-ImportObject.Name)
  TABLE(S-ImportObject.Name) :=
    ModuleTable(S-ImportObject.Name)
else
  Decl := ModuleTable(S-Ident.Name)
  TABLE(S-ImportObject.Name) :=
    ModuleTable(S-Ident.Name)
endif

condition TABLE(S-ImportObject.Name) = undef
and
  (NoNode(S-Ident) **implies** ModuleTable(S-ImportObject) $\neq$ undef)
and
  (**not** NoNode(S-Ident) **implies** ModuleTable(S-Ident) $\neq$ undef)

ImportObject:
  Decl.Return := NextTask
  CurrentTask := Decl

Montage $\mathcal{O}_5$.M.3: Semantics of an import.

The static semantics of Oberon guarantees that the local names of imported modules cannot be used for user defined constants, types, variables, and procedures. Therefore we need to add a static semantics condition for all of them.

CONJUNCT CONDITION OF CONSTDECLARATION $\mathcal{O}_2$.M.7 WITH: ($\mathcal{O}_5$.C.1)
  not ModObject(TABLE(S-ConstObject.Name))

CONJUNCT CONDITION OF TYPEDECLARATION $\mathcal{O}_1$.M.5 WITH: ($\mathcal{O}_5$.C.2)
  not ModObject(TABLE(S-TypeObject.Name))

CONJUNCT CONDITION OF VARDECLARATION $\mathcal{O}_1$.M.1 WITH: ($\mathcal{O}_5$.C.3)
  for all v **in list** S-VarObject **holds**
    not ModObject(TABLE(v.Name))

CONJUNCT CONDITION OF PROCOBJECT $\mathcal{O}_3$.M.1 WITH: ($\mathcal{O}_5$.C.4)
  not ModObject(TABLE(Name))

Another static constraint is that only globally declared code objects may be exported. Therefore a code object in the declaration sequence of a procedure declaration is not allowed to be marked with the export "*".

CONJUNCT CONDITION OF PROCDECLARATION $\mathcal{O}_4$.M.1 WITH: ($\mathcal{O}_5$.C.5)
  for all c: CodeObject
    satisfying c CodeObjectOf S-DeclSequence **holds**
      not c.Exported

Before we specify how the imported objects can be referenced, we complete our collection of minor refinements of the static semantics of Oberon with an additional condition for assignments. The assignment of procedures to procedure variables is only allowed for globally declared procedures. From the structure of modules, declaration, and procedure declarations we see that globally declared procedures are located exactly four levels below their module in the derivation tree.

CONJUNCT CONDITION OF ASSIGNMENT $\mathcal{O}_4$.C.12 WITH: ($\mathcal{O}_5$.C.6)
ProcOrForwardObject(S-Expression.VALUE) **implies**
Module(S-Expression.VALUE.Up.Up.Up.Up)

5.2 External References

External type identifiers are an additional synonym of pointable types ($\mathcal{O}_2$.G.9).

PointableType = ExternalTypeIdent | RecordType | ArrayType ($\mathcal{O}_5$.G.3)

A reference to an externally defined type (Montage $\mathcal{O}_5$.M.4) gets the module object using the TABLE and then accesses the type using the field function.

ExternalType $\triangleq$ ($\mathcal{O}_5$.A.1)
TABLE(S1-Ident.Name).Field(S2-Ident.Name)

The static analysis sets the initial and terminal leaves of an external type identifier to the initial and terminal leaves of the referenced type. The static semantics guarantees that the referenced type is defined, e.g. is member of the universe TypeTask. The static semantics guarantees as well that the module of the external type is imported, and that the used type is exported. This second static semantics constraint is implied from the first, since only imported modules are in the TABLE and only exported types are accessible as fields.

ExternalTypeIdent ::= Ident"."Ident
Initial := ExternalType Terminal := ExternalType
condition TypeTask(ExternalType)

Montage $\mathcal{O}_5$.M.4: Semantics of an external type reference.

Result type identifiers (Montage $\mathcal{O}_5$.M.5) can reference external types as well. As already noted in $\mathcal{O}_4$.A.3 the static analysis must set the global function ResultTYPE.

LocalType $\triangleq$ TABLE(S1-Ident.Name) ($\mathcal{O}_5$.A.2)

ResultTypeId ::= **Ident**[“.”**Ident**]

S1-Ident S2-Ident

if NoNode(S2-Ident) **then**
  Initial := LocalType
  Terminal := LocalType
  ResultTYPE := LocalType
else
  Initial := ExternalType
  Terminal := ExternalType
  ResultTYPE := ExternalType
endif

condition (NoNode(S2-Ident) **implies** TypeTask(LocalType))
and
(**not** NoNode(S2-Ident) **implies** TypeTask(ExternalType))

Montage $\mathcal{O}_5$.M.5: Semantics of a result type identifier.

External constants, variables, and procedures are referenced with qualified designators (Montage $\mathcal{O}_1$.M.13). A qualified designator is an external reference if the type of the argument is a ModObject (see update StaticType := Self in Montage $\mathcal{O}_5$.M.2). In this case the following rule modifies the field selector such that its dynamic behavior corresponds exactly to that of a simple designator.

BLOCK STATIC ANALYSIS OF QUALIFIED $\mathcal{O}_1$.M.13 WITH: ($\mathcal{O}_5$.A.3)
if ModObject(ArgType) **then**
  S-FieldSelector.StaticType := DirectField.StaticType
  S-FieldSelector.Decl := DirectField
  Simple(FieldSelector) := true
endif

The static semantics of a qualified designator must be redefined in order to guarantee that only exported objects are referenced. The first two parts of the following condition guarantee that one accesses only exported fields of externally declared record types. A record type is declared externally if its parent module ParentModule is not equal to the currently analyzed module CurModule ($\mathcal{O}_5$.Σ.3). The third part of the condition guarantees that only exported external objects are referenced.

REPLACE CONDITION OF QUALIFIED $\mathcal{O}_1$.M.13 WITH: ($\mathcal{O}_5$.C.7)
RECORD(ArgType) **and** DirectField $\neq$ undef **and**
(ArgType.ParentModule $\neq$ CurModule
  implies DirectField.Exported)
or POINTER(ArgType) **and** IndirectField $\neq$ undef **and**
(ArgType.PointedType.ParentModule $\neq$ CurModule
  implies IndirectField.Exported)
or ModObject(ArgType) **and**
not TypeTask(DirectField) **and**
DirectField.Exported

5.3 Record Extension

In this section we present record type extensions [8]. A record type extension adds new fields to a given base type. The extension is a sub-type of its base type, e.g. wherever an instance of the base type can be used, an instance of the extended type is allowed as well. If the base type is an external defined type, the non–exported fields may be overwritten. This specific situation is the only possibility to overwrite fields in Oberon, but it leads to a much more general record model. This more general model can be used to model the class concept of strong–typed object–oriented languages like Java.

Syntactically the base type of a record type is given in parentheses after the RECORD-token.

INFIX IN PRODUCTION RECORDTYPE $\mathcal{O}_1$.M.10 ($\mathcal{O}_5$.G.4)
AFTER RECORD:

		["(" BaseTypeId ")"]
BaseTypeId	=	TypeIdent | ExternalTypeIdent

As we have seen for the qualified designators, one needs to know in which module a record type is declared. Thus the static analysis of a record type sets the field ParentModule to the current module.

BLOCK STATIC ANALYSIS OF RECORDTYPE $\mathcal{O}_1$.M.10 WITH: ($\mathcal{O}_5$.A.4)
S-RECORD.ParentModule := CurModule

The field

BaseType: RECORD $\rightarrow$ Type ($\mathcal{O}_5$.Σ.5)

is defined graphically with a data flow arrow from the RECORD-token to a block representing the BaseTypeId. The static semantics must guarantee that the names of the record fields are different from the names of the fields of the base type. An exception are non-exported fields of the base type which may be overwritten, if they externally declared. The condition uses the macro $\mathcal{O}_1$.C.3 in order to quantify over the field of two record types. The term

Self.S-BaseTypeId.Terminal.Up

refers to the RecordType-Node of the base type.

CONJUNCT CONDITION OF RECORDTYPE $\mathcal{O}_1$.M.10 WITH: ($\mathcal{O}_5$.C.8)
not NoNode(S-BaseTypeId) **implies**
RECORD(S-BaseTypeId) **and**
ForFieldObjects f1 Of Self.S-BaseTypeId.Terminal.Up
AndForFieldObjects f2 Of Self
(S-BaseTypeId.ParentModule = CurModule
implies f1.Name $\neq$ f2.Name)
and
(f1.Exported
implies f1.Name $\neq$ f2.Name)

The create service of a record type with base type creates a record y of the base type and then a second new element x is allocated and inherits all fields of y. The two objects x and y are linked by the infix function

$$\text{As: Composite} \times \mathsf{RECORD} \rightarrow \text{Composite} \qquad (\mathcal{O}_5.\Sigma.6)$$

e.g. if the type of y is Y then *(x As Y)* is set to y. The *RecordEnd*-token maintains the reflexive, transitive closure of these links. In the following we give the formal rules for this process.

RECORD: ($\mathcal{O}_5$.D.1)

```
if Mode = create then
  if NoNode(BaseType) then
    extend Composite with o
      Dest := o
    endextend
    CurrentTask := NextTask
  else
    BaseType.CREATE() ;
      extend Composite with o
        Dest := o
        vary s over String
          o.Field(s) := NewObject.Field(s)
        endvary
        (o As BaseType) := NewObject
      endextend
      CurrentTask := NextTask
  endif
endif
```

If there is no base type, the rule is the same as the create service in the old record type Montage $\mathcal{O}_1$.M.10. Otherwise the create service of the base type is called, and afterwards a new object is allocated, the fields of the base type object are copied, and the relation As is updated. Like this there is only a As-linked list from the record to its super–types. The RecordEnd-token of each record type makes the local transitive closure, e.g. defines the As-relation between itself and all its super-types. The relation

$$\text{LocalAsClosure: } \mathsf{RECORD} \times \text{Composite} \rightarrow \text{Boolean} \qquad (\mathcal{O}_5.\Sigma.7)$$

contains all pairs (t, r) such that the new record Parent.Dest interpreted as t is r:

- LocalAsClosure(Parent, Parent.Dest)
- **for** t, r1, r2:
  t ∈ REORD **and** r1, r2 ∈ Composite
  and LocalAsClosure(t, r1)
  and (r1 As t.BaseType) = r2:
  LocalAsClosure(t.BaseType, r2)

Using this relation the dynamic semantics of the RecordEnd-token is

```
RecordEnd:                                                    (O5.D.2)
   if Mode = create then
     vary t over RECORD
       vary r over Composite
         satisfying LocalAsClosure(t,r)
           (Parent.Dest As t) := r
           (r As Parent) := Parent.Dest
       endvary
     endvary
     CurrentTask := Parent.Return
```

The copy service of a record type replaces the source and the destination with their correct versions. If there is a base type, its copy service is used to copy the fields belonging to the base type.

```
RECORD:                                                       (O5.D.3)
   if Mode = copy then
     Scr := (Scr As CurrentTask)
     Dest := (Dest As CurrentTask)
     if NoNode(BaseType) then
       CurrentTask := NextTask
     else
       BaseType.COPY(Src, Dest)
     endif
   endif
```

The changes of the record model triggers the following refinements:

– The macro Assignable to must be refined in order to allow assignments of sub-type instances to super-type instances:

  DISJUNCT DEFINITION OF ASSIGNABLETO $\mathcal{O}_4$.C.11 WITH: ($\mathcal{O}_5$.C.9)
  t1 ExtensionOf *t2*

  where ExtensionOf is recursively defined as follows:

```
t1 ExtensionOf t2   ≜                                         (O5.C.10)
   t1.BaseType = t2 or
   exists t such that
     t1.BaseType = t and t ExtensionOf t2
```

  In combination with the existing definition this implies as well, that a pointer to sub–type instance and a pointer to a super–type instance are assignable.
– In the dynamic semantics of an assignment we have to make a case distinction between pointable (structured), pointer, and ground types. The copy service guarantees type correctness in the case of pointable types, in the case of pointer types, we apply the As–function, and in the case of ground types there is no problem.

REPLACE DYNAMIC SEMANTICS OF ASSIGNMENT $\mathcal{O}_2$.D.3 WITH: ($\mathcal{O}_5$.D.4)

```
":="
  if PointableTypeTask(Left.StaticType) then
    Left.StaticType.COPY(Right.AssocOBJ, Left.AssocOBJ)
  elseif POINTER(Left.StaticType) then
    Left.AssocOBJ.Value :=
      (Right.VALUE As Left.StaticType.PointedType)
  else
    Left.AssocOBJ.Value := Right.VALUE
  endif
```

- In section 4.2 we specified that a variable parameter may only be actualized with a parameter of identical type (see $\mathcal{O}_4$.C.13. This rule is now relaxed for extensions of the parameter type.

VariableParamCompatibleTypes (*t1*, *t2*) $\triangleq$ ($\mathcal{O}_5$.C.11)
t1 = *t2* **or**
IdenticProcTypes(*t1*, *t2*) **or**
t1 ExtensionOf *t2*

5.4 Type Guards and Tests

In this chapter we complete the specification of Oberon with constructs that are needed for strong–typing in a language featuring type extensions: type guards and type tests

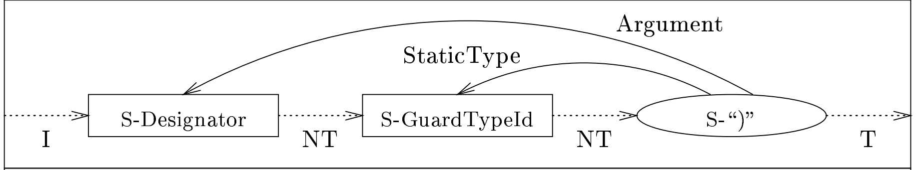
Both type guards and type test consist of a designator and a guarding type. The guarding type must be an extension of the static type of the designator. If the designator references a variable parameter, it may be a pointer or a record, otherwise it must be a pointer. The static semantics of type guard and test is the following macro.

TypeGuardAndTestConditions $\triangleq$ ($\mathcal{O}_5$.C.12)
POINTER(S-GuardTypeId) **and**
S-GuardTypeId.PointedType ExtensionOf
S-Designator.StaticType.PointedType
or
RECORD(S-GuardTypeId) **and**
VariableParamObj(S-Designator.Decl) **and**
S-GuardTypeId ExtensionOf S-Designator.StaticType

The non-terminal TypeGuard is a synonym of Designator. The dynamic semantics of a type guard (Montage $\mathcal{O}_5$.M.6) tests whether the argument can be interpreted as the guarding type. If yes, the corresponding interpretation is assigned to the value respectively the associated object. If no, a run–time–error is raised.

The non-terminal TypeTest is a synonym of Expression. The dynamic semantics of a type test (Montage $\mathcal{O}_5$.M.7) sets the value field to true, if the argument can be interpreted as the guarding type otherwise the value is set to false.

TypeGuard ::= **Designator "(" GuardTypeId ")"**
GuardTypeId = **TypeIdent** | **ExternalTypeId**



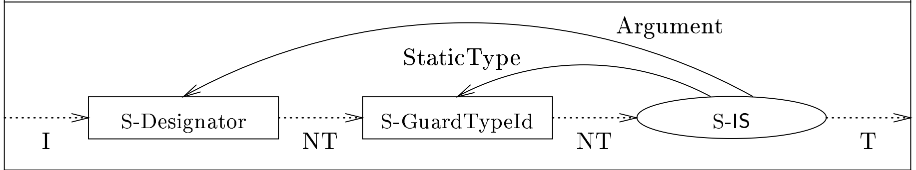
condition TypeGuardAndTestConditions

")":

```
if RECORD(StaticType) then
   if (Argument.AssocOBJ As StaticType) ≠ undef then
      SetTo(Argument.AssocOBJ As StaticType)
      CurrentTask := NextTask
   else
      CurrentTask := RunTimeError
   endif
else
   if (Argument.AssocOBJ.Value As StaticType.PointedType) ≠ undef then
      AssocOBJ:= Argument.AssocOBJ
      VALUE := (Argument.AssocOBJ.Value As StaticType.PointedType)
      CurrentTask := NextTask
   else
      CurrentTask := RunTimeError
   endif
endif
```

Montage $\mathcal{O}_5$.M.6: Semantics of a type guard.

TypeTest ::= **Designator IS GuardTypeId**



condition TypeGuardAndTestConditions

IS:

```
if RECORD(StaticType) and (Argument.AssocOBJ As StaticType) ≠ undef
   or    POINTER(StaticType) and
         (Argument.AssocOBJ.Value As StaticType.PointedType) ≠ undef then
   VALUE := true
else
   VALUE := false
endif
CurrentTask := NextTask
```

Montage $\mathcal{O}_5$.M.7: Semantics of a type test.

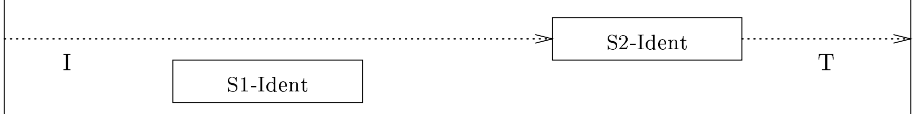
WithStatement	::=	WITH Guard DO StatementSequence WithEnd	($\mathcal{O}_5$.G.5)
WithEnd	=	END	
Guard	::=	VariableId GuardTask GuardTypeId	
GuardTask	=	":"	
VariableId	::=	Ident["."Ident]	

The with statement is used to apply a type guard to a whole statement sequence. The variable identifier is a local or external reference to a variable (Montage $\mathcal{O}_5$.M.8). The static semantics of the guard part of a with statement (Montage $\mathcal{O}_5$.M.9) checks the same as the TypeGuardAndTestConditions.

GuardConditions $\triangleq$ ($\mathcal{O}_5$.C.13)
TypeGuardAndTestConditions with
S-Designator substituted by S-VariableId

The dynamic semantics is similar to the dynamic semantics of a type guard, but it changes directly the associated object of the declaration. Analogously the static analysis changes the typing of the declaration. All changes are reset to the initial values by the with statement (Montage $\mathcal{O}_5$.M.10).

VariableId **::=** **[Ident"."]Ident**

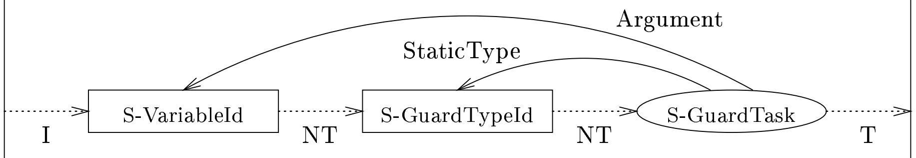



```
Simple(S2-Ident) := true
if NoNode(S1-Ident) then
   S2-Ident.Decl := TABLE(S2-Ident.Name)
   S2-Ident.StaticType := TABLE(S2-Ident.Name).StaticType
else
   S2-Ident.Decl := ExternalVarObject
   S2-Ident.StaticType := ExternalVarObject.StaticType
endif
where ExternalVarObject ≜
      Field(TABLE(S1-Ident.Name), S2-Ident.Name)
```

condition (NoNode(S1-Ident) **implies** VarObject(TABLE(S2-Ident.Name))
and
(**not** NoNode(S1-Ident) **implies** VarObject(ExternalVarObject))

Montage $\mathcal{O}_5$.M.8: Semantics of a variable identifier.

Guard ::= **VariableId GuardTask GuardTypeId**
GuardTask = **":"**



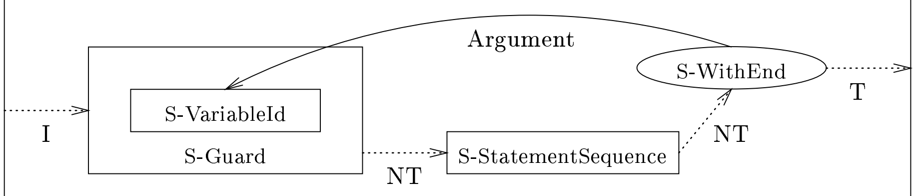
S-VariableId.Decl.StaticType := S-GuardTypeId

condition GuardConditions

GuardTask:
```
if RECORD(StaticType) then
  if (Argument.AssocOBJ As StaticType) ≠ undef then
    Argument.Decl.AssocOBJ:= (Argument.AssocOBJ As StaticType)
    CurrentTask := NextTask
  else    CurrentTask := RunTimeError
  endif
else
  if (Argument.AssocOBJ.Value As StaticType.PointedType) ≠ undef then
    Argument.AssocOBJ.Value :=
        (Argument.AssocOBJ.Value As StaticType.PointedType)
    CurrentTask := NextTask
  else    CurrentTask := RunTimeError
  endif
endif
```

Montage $\mathcal{O}_5$.M.9: Semantics of the guard part of a with statement.

WithStatement ::= **WITH Guard DO StatementSequence WithEnd**
WithEnd = **END**



S-Guard.S-VariableId.Decl.StaticType := S-Guard.S-VariableId.StaticType

WithEnd:
```
if RECORD(Argument.StaticType) then
  Argument.Decl.AssocOBJ:=
    (Argument.Decl.AssocOBJ As Argument.StaticType)
else
  Argument.Decl.AssocOBJ.Value :=
      (Argument.Decl.AssocOBJ.Value As Argument.StaticType.PointedType)
endif
CurrentTask := NextTask
```

Montage $\mathcal{O}_5$.M.10: Semantics of a with statement.

6 Conclusions

In this paper, we gave the specification of all constructs of the programming language Oberon. The features which have not been covered are the built–in procedures, the exact textual representation of numbers and sets, and the module System which defines low level features that break the data abstraction, e.g. allow for byte wise access of the store.

In order to obtain a correct specification we adhered to the original Oberon report [7] and to a book on programming in Oberon [5]. Nevertheless for certain details we had to consult several times the creator of the language [9]. In one case the design-intentions [9] where different to what has been implemented in the original compiler, i.e. a type guard of a pointer to Nil should not raise a run–time–error.

7 Acknowledgments

First of all thanks go to F. Haussmann, who was partner in the modeling and information retrieval process for the dynamic semantics of Oberon. We are indebted to Y. Gurevich for giving us helpful comments on our model and for motivating us in our work during his spring 1995 lectures in Zürich. The treatment of the complete language was only possible with the help of N. Wirth, who answered in detail all our questions concerning Oberon.

References

1. E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *IFIP 13th World Computer Congress, Volume I: Technology/Foundations*, pages 489 – 508. Elsevier, Amsterdam, 1994.
2. Y. Gurevich and J.K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274 – 308. Springer Verlag, 1993.
3. P.W. Kutter. Dynamic semantics of the programming language oberon. Technical Report 25, TIK, ETH Züerich, 1997.
4. P.W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *J.UCS*, 3(5), 1997. This volume.
5. M. Reiser and N. Wirth. *Programming in Oberon*. Addison–Wesley, 1992.
6. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.
7. N. Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7):671 – 690, 1988.
8. N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.
9. N. Wirth. personal communications, emails from feb.1th, feb.5th, nov.28th, 1996.