Dynamic Semantics of the Oberon Programming Language

Philipp W. Kutter, pwkutter@kestrel.edu

December 17, 1996

Abstract

In this paper we present an abstract mathematical model for the dynamic semantics of the *Oberon* programming language using the *Evolving Algebras* approach. Oberon is the object-oriented successor of Pascal and Modula2. The resulting formal specification is complete, compact, and understandable with minimal training.

Contents

1 Introduction

In this paper we present an abstract mathematical model for the dynamic semantics of the object oriented, strongly typed programming language *Oberon* [37, 28]. Following the Evolving Algebras approach, we define an algebraic structure for a fixed but arbitrary Oberon program and model the dynamic semantics of this program as simple imperative code executed over the defined structure. This algebraic structure can be seen as the initial state of the program, which is changed stepwise by the execution of the code. We use the *Evolving Algebras* approach since it provides well founded syntax and semantics for this general type of transition system [11]. Unlike the typical use of ill-defined pseudo code, execution of Evolving Algebras code has a precise, mathematical semantics that corresponds nicely to natural intuition.

Experience shows that Evolving Algebras form a base to verify large and complex systems without being constrained to one specific proof system [6, 5, 7, 8, 9, 12, 34, 16]. For further examples of EA applications, see a bibliography in [3], a survey in [4] and a web page in [33]. Our work takes advantage of the experience gained by these case studies and applies a new methodology specially suited for the formal Evolving Algebras definition of programming languages.

The specification of programming languages is normally presented in two parts, *static* and *dynamic semantics*. The motivation for this is that programs are typically loaded into a computer and then executed. The static semantics specifies which programs can be executed and how the information in the programs can be accessed. The dynamic semantics specifies what happens when the programs are executed. The benefit of the separation of static and dynamic semantics is that both can be based on a very simple representation of the syntax of the programs. This representation is typically a *context-free grammar*. The specification of the dynamic semantics can abstract from most of the static semantics. This separation of concerns allows shorter and more concise specifications of programming languages.

There are two main approaches both in the implementation and specification of programming languages. The first, called *interpretation*, was widely used in the early history of computer science. In this approach, an interpreter program takes the code as input and executes it. The interpreter can be seen as a specification of the dynamic semantics. The static semantics in the interpretation approach is only a definition of which programs may be executed; all information in the code is retrieved at runtime.

The interpretation of high-level programming languages is typically inefficient, which is a major reason why in most cases today an approach called *compilation* is typically used to implement a programming language nowadays. This second approach calculates information statically from the program which will be needed at run time. The result of this calculation is called the *compiled program*. The static semantics in the compilation approach is the specification of how to calculate the compiled program, and the dynamic semantics is the specification of how compiled programs are executed.

Traditionally the theoretical computer science community uses the interpretation approach to specify programming languages. This works as follows: One defines a set of syntactically correct programs using a context-free grammar. The static semantics is then a Boolean-valued function which determines whether a program is statically correct. A variant is to specify the properties of a statically correct program directly. In both cases the set of syntactically correct programs is restricted by the static semantics to the set of statically correct programs. The dynamic semantics is then a function (or specification) which assumes a statically correct program as input. This approach delivers satisfactory results if applied to rather simple functional programming languages.

As soon as more complex, strongly typed imperative languages are involved, the interpretation approach is ill-suited for both the implementation and the specification. This may explain partly the preference of the theory community for pure functional languages that can be nicely described and manipulated within the traditional approach. On the practical side however, the software engineers prefer working with imperative programming languages. While the theory community claims that imperative programming languages have overly complex formal semantics, it seems to be much easier for the average non-academic programmer to model in his mind what an imperative program does than what a functional program does. The traditional approach misses the *natural levels of abstraction* in its description of imperative programming languages.

In the Evolving Algebras approach this gap between theory and practice can be closed. The state-based semantics of Evolving Algebras allows for the use of the compilation approach. It is not a trivial task to present in the compilation approach the formal dynamic semantics of a programming language while abstracting from the formalization of the static semantics. The operational semantics of C [12, 13] was the first paper to show in an elegant way how the compilation approach can be used to define the formal dynamic semantics of a real world programming language while abstracting from the static semantics. In that paper one assumes the information calculated during the verification of the static semantics as given (e.g. one assumes that certain functions denote the declaration and static type of terms designating a variable or object) and stores the results of the dynamic computations in a global state. Although [12] gives no examples, it is illuminating how easily one can analyze the dynamics of arbitrary complex C programs, assuming all informations about control flow, typing and declarations.

This type of short, concise definition of the dynamics of program constructs is especially useful for a programming language like C which provides very low level constructs used for efficiency and consequently guarantees a poor abstraction of the store. In C the store must be modeled as a sequence of bytes. In our work with Oberon it turned out that a formal description of the syntax and static semantics is more appropriate whenever a language guarantees a more abstract model of the store. The dynamic semantics of Oberon allows for instance to model the store as a set of abstract objects like bounded arrays and records. If the store is modeled on this abstraction level, one needs additional information during runtime, in order to satisfy the consistency constraints of the store. The formal description of the syntax and static semantics helps to collect this additional information.

The consistency of complex store models, commonly called *strong typing*, is an essential feature of high-level object-oriented programming languages like Oberon, Eiffel or Java. We mention Java because its recent success indicates that the software industry is now adapting program language concepts like strong typing proposed by the academic community for decades. We mention Eiffel, because the definition of its dynamic semantics [1] in Kahn's Natural Semantics [17] shows clearly the deficiency of the traditional approach for these programming languages. The paper [1] shows that it is possible to calculate all information at runtime from the original program text and provides even an executable version of the semantics using the Centaur system [10]. The problem is that the rules are too long and hard to read, and that the execution time for longer programs explodes.

We provide a methodology that can be used to specify complex real world programming languages using the compilation approach. The method is based on a minimum of theory, namely context-free grammars and Evolving Algebras; rules are short and readable, furthermore they are executable. We describe the syntax of Oberon using the *extended Bacchus-Nauer form (EBNF)*, which is a widely used formalism to describe context free grammars. The programs generated by the grammar can be seen as *derivation trees*. We use a variant of them called *compact derivation trees* [27]. The compact derivation tree of a program is fully defined by the grammar rules. The definition of an abstract syntax tree grammar is superfluous. We interpret the compact derivation tree as an algebraic structure and define static and dynamic semantics by an Evolving Algebra which takes this structure as initial state. The part of the Evolving Algebra that reflects Static semantics of a program P can be seen as an abstract algorithm performed on the compact derivation tree of P. It traverses the tree, checks the rules and encodes the information needed by the dynamic semantics by means of functions connecting the leaves of the tree. The leaves of the tree together with the functions defined on them are sufficient to execute the part of the Evolving Algebra that reflects the dynamic semantics. As the leaves all represent some syntactic token in the program, the part of the Evolving Algebra that reflects the dynamic semantics can be seen as an algorithm that works on the program text and not on the whole syntax tree. This paper concentrates on the formal dynamic semantics of Oberon. We use the compact derivation trees only in order to define informally the functions that provide the control flow, declarations, and static type information. A formal definition of the static semantics of Oberon using this approach plus a semi-graphical language, called Montages [21] is given in [20]. As it stands, this paper defines formally the dynamic semantics of Oberon and abstracts from the details of the static semantics not used at runtime. We claim that this is the natural abstraction level to define the dynamic semantics of a programming language. The work presented here is self-contained, and [20] is not needed to understand it.

Section 1.1 gives a brief overview on the history of the problem. Section 1.2 gives a simple description of Sequential Evolving Algebras, omitting the algebraic terminology of [11]. Section 2 explains how the EBNF rules give rise to carrier sets and static functions of the initial state. Section 3 gives an overview of the dynamics. The main part of the paper is section 4, where we define for each construct in the language a piece of pseudo-code modeling its dynamic behavior. Interleaved with the pseudo code descriptions are prose descriptions of the respective parts of the initial state.

1.1 Historical View

It is a well-known challenge to specify the syntax also the semantics of programming languages and many people tried to do that. In [15] N. Wirth and C.A.R. Hoare specify the semantics of the programming language Pascal [36]. The axiomatic approach [14] was considered to be the best suited formalism at the time. But the specification is not as easy to understand as hoped, and side effects of procedures cannot be specified directly.

About twenty years later, Yuri Gurevich developed a formalism called Evolving Algebras or ealgebras [11]. With this formalism one can describe the operational semantics of programming languages (for example, C [12]) in a very short and comprehensive way. Only sets and functions are used in the Evolving Algebras methodology to define a model with the same behavior as the language to specify. This makes the resulting models understandable for engineers. Despite their simplicity, ealgebras are completely formal and even executable.

The strict separation between model-theoretic aspects and proof-theoretic aspects is one of the reasons why the Evolving Algebras approach gives a short and comprehensive specification of a real-world programming language. The axiomatic semantic approach of C.A.R.Hoare is specially tailored for a specific proof calculus that is even used to give the specifications. The same is true for most other approaches, like denotational semantics [31] [23] or natural semantics [17] [18]. The Evolving Algebras approach, which is different than the others, leaves all proof-theoretic aspects open and concentrates on the model-theoretic aspects.

We give the formal operational semantics of the programming language Oberon [28] using the Evolving Algebras methodology. Oberon is a direct descendent of Pascal and of Modula2 that supports object oriented programming. Our work has the same purpose as that of C.A.R.Hoare and N.Wirth in [15]: To give a machine-independent mathematical model of the semantics of a programming language, upon which programmers and compiler developers can rely. This is the necessary basis for truly portable software, unambiguous formulations of algorithms as programs and proofs of properties of programs.

1.2 Sequential Evolving Algebras

The Evolving Algebras approach has been used for many purposes [3]. We use only the part of the method called Sequential Evolving Algebras, which we describe below. For a full treatment of ealgebras see [11].

Evolving algebras are versatile abstract machines. An *ealgebra* consists of a set, called the *superuniverse*, and of *functions* with Cartesian products of this superuniverse as domains and the superuniverse as codomain. The machine runs by changing (the definitions of) the functions.

We consider this run to be split in *states* and *transitions*. In a state, all functions are well defined and do not change. A transition changes the functions. The result of a transition is a new state. The functions that never change are called *static*, and the others are called *dynamic*. A *transition rule* specifies how the functions change from one state to another.

The initial state The superuniverse always has distinct elements *true, false* and *undef.* A partial function is modeled by a total function that produces *undef* for certain arguments and we use *undef* as a default value [1]. The usual logical connectives *and, or, not* and *implies* are defined as usual for arguments in { *true, false* }.

Unary functions from the superuniverse to { *true, false* } are called *universes*, because they define by the preimage of *true* a subset of the superuniverse. The universe { *true, false* } is called *Bool.* We use names beginning with a capital letter to refer to a universe and the same name beginning with a lower-case letter to refer to its members (e.g. elements of the universe *Object* are called objects). Members of universes with composed names are called by using their components. Members of VarDeclaration would be called var declarations, or even variable declarations when unambiguous. We write $f : A \rightarrow B$ to indicate that function f produces results in B (or *undef*) for arguments in A and *undef* for any other argument.

For convenience we use a *graph* to represent some of the initial state. Circle-shaped nodes represent distinct elements of the superuniverse, and box-shaped nodes represent parts of the graph that are defined elsewhere. A *directed edge* or *arrow* with label f from a node representing a to a node representing b indicates that the unary function f maps element a to element b. To describe this situation, we say as well that *field* f of a is set to b, and we normally use the notation $a.f$ instead of $f(a)$. An arrow with label $g(e)$ from a node representing a to one representing b , where e is an element of the superuniverse, defines the binary function g to map (a, e) to b. We use *round arrows* for static functions and *straight arrows* for dynamic functions.

A language to specify transition rules A transition rule defines how functions change from one state to the next. The language of transition rules is built recursively upon a number of constructs, which may use a vocabulary of terms evaluated over the current state. The constructs are the following.

- The *update* construct modifies a function in a pointwise manner. For instance, the update

  $$funct(term_1, ..., term_n) := term_0$$

  evaluates first the terms $term_0$, $term_1$, ..., $term_n$ in the current state to the elements e_0, e_1, ..., e_n and then redefines function $funct$ at the point (e_1, ..., e_n) to e_0.

  We will be loose in the informal descriptions of updates and use for the update f(x):= y descriptions ranging from "we update f at point x to y" to "element y is assigned to field x of f".

- The *block* construct allows a set of transition rules to be executed in parallel. The transition rules in the block appear in an arbitrary order.

- The *conditional* construct is a guarded rule of the following form

  if *cond* **then**
    R
  endif

[1] If we do not define the result of some function for some argument, the result is by default defined to be undef

where *cond* is a term and R is a transition rule. R is triggered if and only if *cond* evaluates to *true*. The rule **if** *cond* **then** *R1* **else** *R2* **endif** is an abbreviation of a block consisting of the following conditional rules: **if** *cond* **then** *R1* **endif** and **if** $\neg$ *cond* **then** *R2* **endif**.

- The construct *extend* extends the current state with a new element. The transition rule

  extend Universe **with** e
  R
  endextend

  adds a completely new element to the universe *Universe* and executes transition rule R. Adding new elements means providing elements which are not in the range of any function. Access to the new element within transition rule R is provided by the name e.

We use higher-order updates(e.g. the update of the curried version of a binary function $f : A \times B \rightarrow C$ on a point in A: f(a1) := f(a2)), but only if they can be unambiguously and finitely translated in normal updates.

For simplicity, we avoid external functions in this paper(details about external functions can be found in [11]). In fact, external functions would make certain aspects more natural, but the differences appear only in situations that are considered *bad programming style* by the language designers of Oberon. We will not state the provocative thesis that we do not want to punish good programmers with complications needed only for bad code. Rather, we keep the semantics simple out of the belief that the more complicate semantics would probably irritate more than help.

1.3 Acknowledgments

First of all thanks go to F. Haussmann, who was partner in the modeling and information retrieval process for the Oberon EA. We are indebted to Prof. Y. Gurevich for giving us helpful comments on our model and text and for motivating us in our work. His lectures in Zürich were a source of inspiration. The treatment of the complete language was only possible with the help of Prof. N. Wirth, who answered in detail all our questions concerning Oberon. Thanks to the reading and critical comments of Dr. C. Gomes, S. Ludwig, A. Pierantonio, C.R. Wallace and Prof. A.V. Zamulin. We would like to thank Prof. E. Engeler, who made it possible for us to do this research at ETH Zürich, and the Kestrel Institute where we finished the writing.

2 From Syntax to Structures

The definition of valid programs typically contains *context-free* and *context-sensitive* rules. The usual way to split up this definition is in two steps: first a superset of the valid programs is defined by a *context-free grammar* expressing all context-free rules. Then this superset is reduced to the set of programs satisfying all context-sensitive rules. The definition of this set of valid programs is called the *static semantics* of a programming language.

EBNF rules are a well suited and common tool to do the first step of the static semantics definition. The second step is usually done in prose or by rather complex formalisms [27]. In [21] we develop a graphical language Montages for this task, using Evolving Algebras. In the rest of the paper we consider only valid programs. A formal definition of the static semantics of Oberon can be found in [20] and [27]. For a prose approach we refer to [37] and [28].

This text differs from others concentrating on dynamic semantics (e.g. [1]) by reusing the properties of the static semantics. Examples of such properties in the case of Oberon are strong typing and the obligation to declare identifiers. We can use these properties to provide direct access to the types and the declarations of identifiers.

In subsection 2.2 it is shown how this information is provided, based on the universes and selector functions defined in subsection 2.1. Subsection 2.3 explains how control and data flow information is incorporated in the same structures. In section 4 the details for each language construct are given, interleaved with the definitions of the dynamics of the language constructs.

2.1 Definition of the Universes and of Selector Functions

The starting point of our formalization is the *context-free syntax* of Oberon. This syntax is defined by a *context-free grammar* G_O defined over a vocabulary Σ_O consisting of *terminal* and *nonterminal symbols* (see appendix B for the whole grammar). The grammar we give is an adaptation of the one given in [27] and produces the same strings as the grammar in [28]. Certain details of [27] are omitted in order to make the presentation shorter. One distinguished nonterminal is the *start symbol*, which we call *Program*.

The grammar consists of *EBNF production rules* of the form

$$n ::= E$$

They can be interpreted as *rewriting rules* specifying which strings over Σ_O can replace nonterminal n. These strings are specified by the regular expression E as follows.Let E_1 and E_2 be arbitrary regular expressions over Σ_O.

- The expression consisting of a symbol s specifies the set containing only a string s.
- The expression $[E_1]$ specifies the set of strings in E_1 together with the empty string.
- The expression $E_1|E_2$ specifies the union of the strings in E_1 and the strings in E_2.
- The expression $E_1 E_2$ specifies the set of all possible concatenations of a string in E_1 followed by a string in E_2.
- The expression $\{E_1\}$ specifies the set of strings obtained by concatenation of zero or more strings in E_1.

The terminal symbols of G_O are defined as those which do not occur at the left side of any production rule. The language defined by G_O is the set of terminal-symbol strings derived by starting with the start symbol and by applying the production rules to it; the production rules are applied to the produced string of the last application, as long as there are nonterminals. An application of a production rule $n ::= E$ to a string S replaces some occurrence of n in S with a string specified by E.

Derivation trees The generation of a string S can be described by a *derivation tree*. Leaf nodes are labeled with terminals, and interior nodes are labeled with nonterminals. A node is not uniquely determined by its label but by its position in the tree.

Construction of the tree starts with its root, which is labeled with the starting symbol. For every replacement of a nonterminal n by string $s_1\ s_2\ ...\ s_m$ in the derivation of S we append m nodes, representing s_1 through s_m, to the node x representing n. Node x is always labeled with the left-hand nonterminal n. The right-hand nodes are labeled $s_1, s_2, ..., s_m$ from left to right.

Compact derivation trees Our derivation trees may have chains of nodes with only one child. To make our derivation trees more compact, we contract all nodes in such a chain to one node having all labels of the nodes in the chain. We augment our context-free grammar with productions of the form

$$n = E$$

These productions are called *synonym productions.* The right-hand side of a synonym is restricted to be a single symbol or an alternative $s_1|s_2|...|s_m$ consisting only of single symbols. A grammar containing synonym productions generates the same language as the grammar obtained from it by replacing every "=" meta symbol with "::=". While constructing the tree, we do treat the replacement of n by string s_1 differently if it is specified by a synonym production [2]. Instead of appending a node labeled s_1 to the node x representing the nonterminal to replace, we add an additional label s_1 to x. The synonym productions can thus be used to contract the chains in the derivation trees.

Universes We define the basic universes of our model directly on the compact derivation trees. As mentioned earlier we consider a fixed but arbitrary Oberon program. The universe *Node* contains all nodes of the compact derivation tree of that program. Each symbol s in Σ_O is interpreted with a universe containing all elements in *Node* that are labeled with s. We do not distinguish between terminal and nonterminal symbols for this definition. As usual we use the same name for the universe and the symbol.

Selector functions We define functions that provide access to the direct descendants of nodes. The definition is chosen to be as simple and natural as possible. We give a straightforward definition depending on the form of the involved production rule. For the following definitions we assume there is a node x whose descendants have been constructed by a replacement P in the derivation, and we assume P is specified by a production rule $n ::= E$.

- If E is of the form $s_1 s_2 ... s_m$ we access the nodes representing s_1 through s_m by a family of static unary functions $(\text{S-}s_i : \mathit{Node} \to s_i)_{i \in \{1,\ldots,m\}}$. This works only if the symbols are all distinct. If a symbol s occurs more than once in the sequence $s_1 s_2 ... s_m$, we enumerate the functions, accessing them from left to right: S1-s maps x to the first s-descendant, S2-s to the second, and so on.
- If E is of the form $E_1|E_2$, we do not specify any selector functions.
- If E contains a symbol in a [ ] part, then the selector functions are defined as in the case without [ ], but there can also be an element of the universe *NoNode* as result.
- If E contains lists we are using an intuitive notion. A formalization would be straightforward.

Example As an example we give the grammar of Oberon expressions. The start symbol of this part of G_O is Expression and the terminals are intnumber, TypeIdent, the keywords IN, IS, OR, DIV, MOD and NIL given in Oberon font and the symbols given between "". Note that the "" are not part of the symbols. In the whole grammar G_O nonterminal Factor will have more alternatives and TypeIdent will be a nonterminal.

<table>
<tr><td>Expression</td><td>=</td><td>SimpleExpression | Relation | TypeTest</td></tr>
<tr><td>Relation</td><td>::=</td><td>SimpleExpression RelOp SimpleExpression</td></tr>
<tr><td>RelOp</td><td>=</td><td>"=" | "#" | "<" | "<=" | ">" | ">=" | IN</td></tr>
<tr><td>TypeTest</td><td>::=</td><td>SimpleExpression IS TypeIdent</td></tr>
<tr><td>SimpleExpression</td><td>=</td><td>Term | SignedTerm | Sum</td></tr>
<tr><td>SignedTerm</td><td>::=</td><td>Sign Term</td></tr>
<tr><td>Sign</td><td>=</td><td>"+" | "-"</td></tr>
<tr><td>Sum</td><td>::=</td><td>SimpleExpression AddOp Term</td></tr>
<tr><td>AddOp</td><td>=</td><td>"+" | "-" | OR</td></tr>
</table>

[2] Note that the restricted form of the right-hand expression in a synonym production guarantees that this string consists of a single symbol.

Universe	Members
Expression	$1, 6, 24$
Relation	6
RelOp	12
TypeTest	
SimpleExpression	$1, 2, 11, 13,$ $14, 20, 24, 26$
SignedTerm	
Sign	
Sum	$1, 11, 14, 24$
AddOp	$3, 15, 21, 27$
Term	$2, 4, 8, 13,$ $16, 17, 20, 22,$ $26, 28$
Product	$4, 13$
MulOp	$9, 18$
Factor	$2, 8, 10, 16,$ $17, 19, 20, 22,$ $26, 28$
ExprInParentheses	$2, 19$
"("	$5, 23$
"+"	$15, 21$
intnumber	$16, 17, 20, 22,$ $26, 28$
TRUE	8
⋮	⋮

Table 1: Members of universes defined by the compact derivation tree in figure 1

Term	=	Factor | Product
Product	::=	Term MulOp Factor
MulOp	=	"*" | "/" | DIV | MOD | "&"
Factor	=	ExprInParentheses | intnumber | TRUE | FALSE | NIL
ExprInParentheses	::=	"(" Expression ")"

We assume for this example a fixed Oberon expression. Let us consider:

(3 + 5 + 7 < 8 * (3 - 2)) OR TRUE & FALSE

The compact derivation tree of this string is pictured in figure 1.

The compact derivation tree in figure 1 defines the universes defined in table 1 and the selector functions in table 2.

Lexical representation The atomic elements of the context-free languages are the terminal symbols. There are two classes of terminal symbols. Those with a fixed lexical representation, for instance "<" or FALSE, and those with a variable lexical representation, for instance intnumber. The lexical representation of terminals of the second class is as well called their *micro syntax*. The micro-syntax of terminals can be accessed using the function *ConstVal*. For the sake of simplicity, we assume that ConstVal produces directly the mathematical denotation of the micro syntax, instead of the micro syntax itself. The definition of ConstVal trough the compact derivation tree in figure 1 is $16 \mapsto 7$, $17 \mapsto 8$, $20 \mapsto 3$, $22 \mapsto 5$, $26 \mapsto 3$, $28 \mapsto 2$. Please note that the numbers

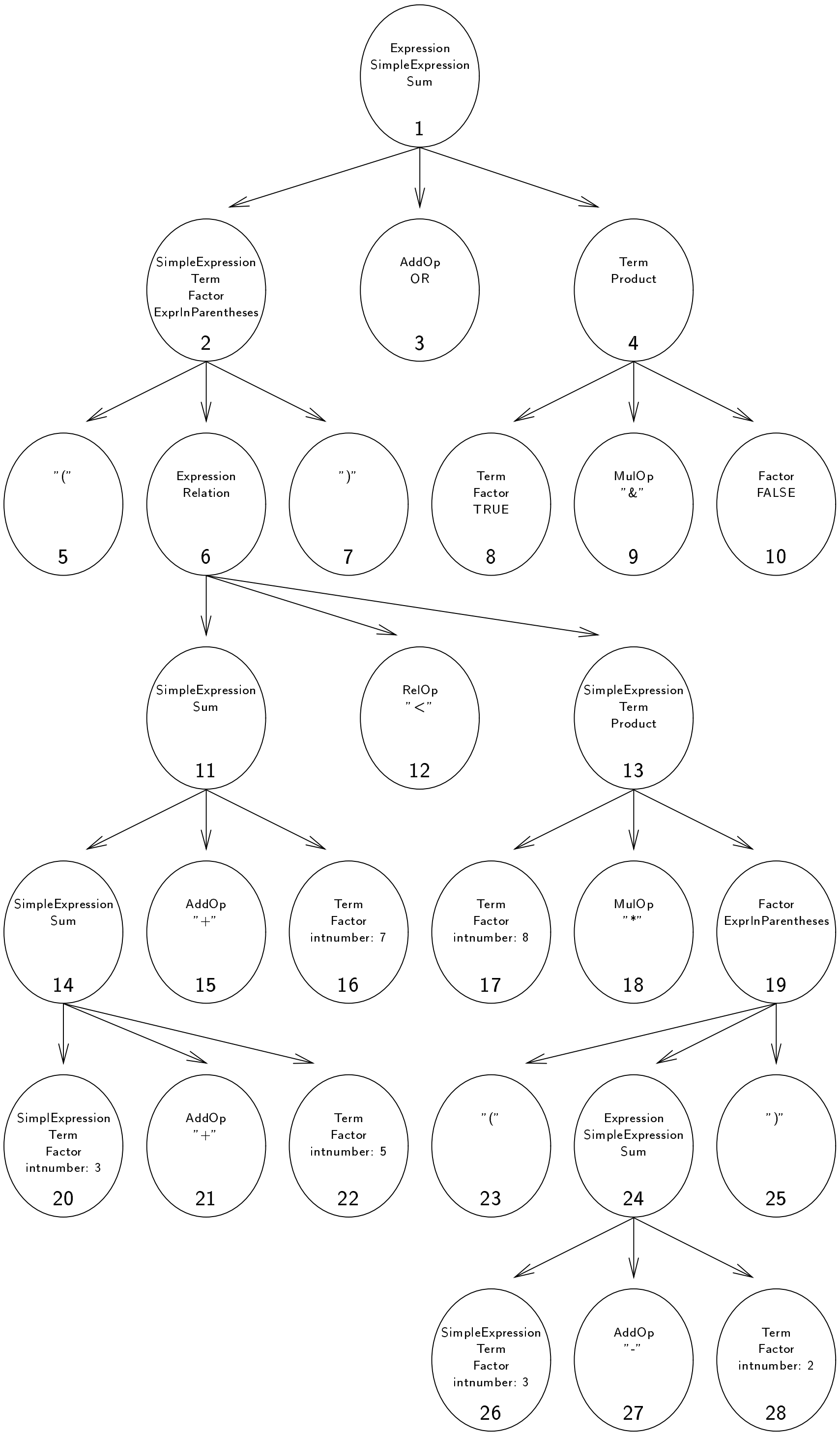

Figure 1: Compact derivation tree of (3 + 5 + 7 < 8 ∗ (3 - 2)) OR TRUE & FALSE

Selector function	Definition
S-Expression	$2 \mapsto 6, 19 \mapsto 24$
S-Relation	
S-RelOp	$6 \mapsto 12$
S-TypeTest	
S-SimpleExpression	$1 \mapsto 2, 6 \mapsto 11,$ $6 \mapsto 13, 11 \mapsto 14,$ $14 \mapsto 20, 24 \mapsto 26$
S-SignedTerm	
S-Sign	
S-Sum	
S-AddOp	$1 \mapsto 3, 11 \mapsto 15,$ $14 \mapsto 21, 24 \mapsto 27$
S-Term	$1 \mapsto 4, 4 \mapsto 8,$ $11 \mapsto 16, 13 \mapsto 17,$ $14 \mapsto 22, 24 \mapsto 28$
S-Product	
S-MulOp	$4 \mapsto 9, 13 \mapsto 18$
S-Factor	$4 \mapsto 10, 13 \mapsto 19$
S-ExprInParentheses	
S-“(”	$2 \mapsto 5, 19 \mapsto 23$
S-“+”	
S-intnumber	
S-TRUE	
⋮	⋮

Table 2: defined by the compact derivation tree in figure 1

in the domain of the mapping ConstVal are node numbers of figure 1, and the numbers in the codomain are integers.

2.2 Declarations and Typing

This section describes the context-sensitive rules of the static semantics and how the information extracted from them is stored in the functions *Decl* and *Type*. Functions *Decl* and *Type* are used in the dynamic semantics in order to access the declaration and type of an element in the code directly.

Function Decl A context-sensitive rule guarantees that the identifiers denoting constants, types, variables (including parameters) and procedures are declared somewhere. Therefore we can use in the dynamic semantics a function *Decl* that maps those identifiers to their declarations. Identifiers are nodes labeled with *ident*. The declarations of these identifiers are labeled with *CodeObject* and occur in *DclSequence*-subtrees, so-called declaration sequences.

DclSequence	::=	[CONST {ConstDeclaration ";"}]
		[TYPE {TypeDeclaration ";"}]
		[VAR {VarDeclaration ";"}]
		{ProcDeclaration ";" | ForwardDeclaration ";"}
ConstDeclaration	::=	ConstObject "=" ConstExpression
ConstObject	=	CodeObject
ConstExpression	=	Expression
TypeDeclaration	::=	TypeObject "=" Type
TypeObject	=	CodeObject
VarDeclaration	::=	VarObject { "," VarObject } ":" Type
VarObject	=	CodeObject
ProcDeclaration	::=	PROCEDURE ProcObject ProcedureBlock ident
ForwardDeclaration	::=	PROCEDURE "↑" ProcObject ParamBlock
ProcObject	=	CodeObject
FPSection	::=	VAR VarParamObject { "," VarParamObject } ":" FormalType
		| ValParamObject { "," ValParamObject } ":" FormalType
VarParamObject	=	CodeObject
ValParamObject	=	CodeObject
CodeObject	=	ident

Decl links an identifier to the (syntactically) first preceding code object having the same micro syntax. The micro syntax of a nonterminal *ident* is accessed by the function

$$Name : ident \rightarrow String$$

One exception to the above definition of the *Decl*-function takes place when the preceding code object with the same micro syntax occurs in a forward declaration. In this case *Decl* maps the identifier to the first (syntactically) following procedure object having the same syntax match (thus to the real procedure declaration instead of the forward declaration). Another exception is a predefined procedure. Such procedures are represented by predefined code objects, so-called *StdDeclarations*, and the *Decl*-function of an identifier having the same name as a predefined procedure object is set to the corresponding code object. We do not treat all predefined procedures in this paper. If they are simulated with ordinary procedures, this can be done statically and is thus deferred future work.

There are identifiers not referencing a code object, so their *Decl*-fields remain *undef.* An example is a reference to a field of a record.

In a valid program, identifiers denoting constants will be declared by *ConstObjects*, those denoting types will be declared by *TypeObjects*, those denoting variables will be declared by *VarObjects, VarParamObjects* or *ValParamObjects*, and those denoting procedures will be declared by *ProcObjects*. In the compact derivation trees defined by the G_O grammar, all nodes labeled with *CodeObject* are additionally labeled with the name of one of the above mentioned special objects.

Qualified identifiers Instead of identifiers, one can use qualified identifiers to denote constants, types, variables and procedures of other modules.

Qualified	::=	Designator "." FieldSelector
FieldSelector	=	ident

where the designator is an identifier whose micro syntax equals the name of an imported module M. Note that this condition is context-sensitive. In this case the identifier *FieldSelector* after the point "." must be defined in the declaration sequence (*DclSequence*) of a module M, and *Decl* maps the *FieldSelector*-node to this declaration in M.

To identify references to code objects, we define the universe *BasicReference* to contain all idents referencing some code objects.

$$\mathit{BasicReference} = \{n \mid \mathit{CodeObject(Decl(n))}\}$$

Function Type A function called *Type* denotes the type of an identifier. The type of an identifier is defined to be the type of its CodeObject. A CodeObject o is always a descendant of a node having a Type descendant t. This type determines the type of the CodeObject: if t is again a (qualified) identifier, then the type of this identifier is the type of o; else t is the type of o. Identifiers denoting ground types are considered to be types and not references to some predefined type.

2.3 Control and Data Flow

The execution of Oberon programs abstracts from the inner nodes of the tree; the only nodes involved are the leaves of the tree. To model the control and data flow of the execution, we define several functions that connect the leaves. The definition of these functions uses some of the tree structure. The abstraction of the used tree structure consists of the selector functions and the two functions

$$\mathit{Initial:\ Node} \rightarrow \mathit{Node} \qquad\qquad \mathit{Terminal:\ Node} \rightarrow \mathit{Node}$$

The function *Initial* typically matches the root of a subtree to its first leaf in the control flow. We guarantee this for the subtrees generated by the nonterminals Module, Procedure, Statement, Expression (and its specialized versions Relation, Sum, ..), VarDeclaration, Designator and Type.

The function *Terminal* typically matches the root of a subtree to its last leaf in the control flow. The same terminal leaf is used in our model to access the computed values of this subtree. Examples of nonterminals generating subtrees with a terminal leaf are Statement, Expression (and again its specialized versions), Variable and Designator.

For notational convenience, we will use a nonterminal as domain or codomain of a function to denote the possible labels of its terminal leaf. If, for example, we want to express, that the function *Left* has the universes "+", "-", "*" etc. as codomain, we write simply *Left: Node* $\rightarrow$ *Expression*.

The task of the static semantics is to verify the context-sensitive rules and to link the leaves. We are only interested here in the linking of the leaves. With the help of the Initial and Terminal functions we can define this locally for each grammar rule. The rule defines the direct descendants of some node n. These descendants are again roots of some subtrees. We first connect the leaves of all these subtrees and retain the information about the initial and terminal leaves. With this information we can now complete the connection of the leaves of n and define which leaf is the initial and which leaf the terminal one of n.

The definition of the functions Initial and Terminal is thus given inductively; we use their values on the descendants of a node n to define them on n. Let us take the $<$-nodes as example. The initial leaf of a $<$-node is defined to be the initial leaf of the left expression descendant of this node.

These functions are defined ambiguously in the semantics of Oberon, since the order of evaluation is not always defined in Oberon. For example the order of procedure argument evaluation and the order of the initialization of imported modules are not defined. The formalization of all possible orders can be done by means of external functions. In this paper we will take the syntactical left to right order as the order of evaluation which simplifies the specification.

3 Dynamics Overview

In this section we give an overview of the dynamic semantics using the structure defined in the last section. The first subsection shows how the sequential execution is modeled based on the control flow information in section 2.3. The second subsection explains the additional structures used to model the store and introduces the common dynamic behavior of types. The third subsection introduces statements, expressions, variables and designators.

3.1 Sequential Execution

Oberon is a sequential programming language. Therefore we model the execution of a Oberon program as a sequential execution of units of work or *tasks*. Most leaves in the derivation tree represent such units.

We mentioned that an ealgebra runs by changing the definitions of the functions according to the transition rule. A sequential program typically runs by executing one *machine instruction* after the other. The current instruction is pointed by a "program counter". For convenience we model our ealgebra in the same way. The tasks (leaves) are our instructions; a dynamic nullary function

$$CurrentTask : Node$$

is used as "program counter". The *CurrentTask*, abbreviated as *CT*, is said to *have control*. The transition rule does the work of the current task and assigns the next task in the control flow to *CT*. We therefore say we *pass control* to the next task. A task is always a leaf of the compact derivation tree, but there are leaves that are not tasks (e.g. they will never get control). In the rest of the paper we use the terms node, leaf and task interchangeable.

To decide which task is next one in the control flow, the static functions introduced in section 2.3 are used. For most tasks the successor in the control flow can be determined statically from the syntax. This is reflected by the static function

$$NextTask : Node \rightarrow Node$$

that links *tasks* sequentially. We abbreviate this function as NT. If we want to link with the NT-function a set of nodes to a list, we express this by saying that those nodes are linked to a NT-list in a certain order.

Typically the NT-function will be defined so that it links the terminal node of one subtree with the initial of the next. This linking is one part of the definition of the initial state, given in section 4.

If a runtime error occurs (e.g. division by zero, range overflow or dereferencing of a NIL-pointer), control is passed to a distinct element *Trap*. All exceptions are caught by the Oberon language, and the program is guaranteed to terminate in a defined state, unless it encounters a non terminating loop.

3.2 Objects and Types

We model the store as a growing universe *Object*, which is partitioned into *Location*, *Record* and *Array*.

$$Object = Location \cup Record \cup Array$$

The universe *Type* contains all nodes of the derivation tree labeled with *Type*.

Type	=	TypeIdent | GroundType | PointerType | ProcedureType | RecordType | ArrayType

The universe *Unstructured* of *unstructured types* is a subset of *Type*, defined as follows:

$$Unstructured = PointerType \cup ProcedureType \cup GroundType$$

The *typing* of objects is modeled by a binary relation

$$TypingRelation : Object \times Type \rightarrow Bool$$

We abbreviate it in formulas with *TR*. Once an object has been created it does not change its typing during execution. If $TR(o,t)$ then we say that o is of type t or that o is an object of t.

Locations A location is an object of an unstructured type, and its information is stored in a dynamic field *Value* (see section 3.3 for expressions and designators).

$$Value : (Location \cup Expression \cup Designator) \rightarrow OberonValue$$

where *OberonValue* consists of the values of the Oberon ground types, the first tasks of all procedures (see 4.1), Record, Array, and the element *Nil*. The values of the basic types are the union of *Longreal*, *Set* and *Bool*. The following subset relation is valid:

$$Shortint \subseteq Integer \subseteq Longint \subseteq Real \subseteq Longreal$$

The universes *Shortint*, *Integer* and *Longint* are sets of consecutive integers ranging from a minimal (typically negative) value to a maximal(typically positive) value. *Longreal* and *Real* are sets of real numbers with minimal and maximal elements. We abstract from the details of computer real arithmetic by using mathematical real arithmetic, which includes integer arithmetic. Our model is thus not suited for the behavior of complex numeric algorithms.

The universes holding the values of the basic types correspond to the predefined identifiers for Oberon ground types:

GroundType	=	BOOL | SHORTINT | INTEGER | LONGINT | REAL | LONGREAL | SET | CHAR

Records A record is a structured object, which has named and typed fields to store objects of these types. A dynamic function

$$Field : Record \times String \rightarrow Object$$

is used to store this information. Only records of a defined RecordType are allowed. The fields may be shared by two or more records, if their types are in a subtype relation. One says a record field is inherited by the subtype record from the supertype record. Records having common fields are linked by the infix $\_\ As\ \_$ function. See section 4.4 for details.

Arrays An array is an object with a linear structure of typed objects. A dynamic function

$$Index : Array \times Nat \rightarrow Object$$

gives access to the objects.

Typing constraints A type defines *typing constraints* upon its objects. The typing constraint of a location ensures that its value is of appropriate type, and the typing constraints of records and arrays ensure that their fields are of appropriate types. This could be formalized as invariants of the store with respect to the typing relation *TR*. For simplicity and ease of readability we defer formalization elsewhere whenever there is no ambiguity.

During execution of the code the typing constraints of all objects are satisfied. To guarantee this, we model the types as active tasks that create and manipulate the objects without violating the typing constraints. This is maybe the most unusual point of our model: types are not built in the model. They are part of the code like, for instance, statement-nodes and they can get control and execute actions. We guarantee a minimal structure of three dynamic unary functions *Return*, *Src* and *Dest*, which will be applied to the initial node of the subtrees generated by the non-terminal *Type*. For convenience we will speak of the return address, the source and the destination of the type.

Each type T has two different modes of execution which are chosen by the dynamic nullary function

$$Mode : \{copy, create\}$$

In *copy* mode two objects of T-compatible [3] types are assumed to be assigned to the *Src* and *Dest* fields, which are the *source* and the *destination* of the copy action. The task of T is to set the value(s) of the destination to the value(s) of the source. The objects themselves are not changed. The following macro [4] COPY is used in the transition rules as the standard method to use the copy mode: set mode to copy, assign the source and destination to the appropriate fields, set the next task as return, and pass control to the type.

COPY(*type*, *source*, *destination*) $\triangleq$

Mode := copy
type.Src := *source*
type.Dest := *destination*
type.Return := CT.NT
CT := *type*

[3] The static semantics of Oberon guarantee that this compatibility is satisfied for all valid Oberon programs. We can therefore abstract here from a definition of type compatibility.

[4] A macro is a means of abbreviating text: it is textually replaced by its definition, after actualization of the parameters.

In *create* mode, a new object of the type is created. Certain functions are updated, additional objects are created in order to satisfy the typing constraints upon the new object, and the new object is assigned to the *Dest* field.

The macro **CREATE_A** *type* **THEN** *Transition_Rule* creates an object of *type* and then executes *Transition_Rule*. The new object can be accessed in *Transition_Rule* by *type.Dest*. To retain the intermediate state while control is passed to the type, the macro uses the boolean field *Allocated*, which is initialized with false.

CREATE_A *type* **THEN** *Transition_Rule* $\triangleq$

> **if not** CT.Allocated **then**
>   Mode := create
>   *type*.Return := CT
>   CT := *type*
>   CT.Allocated := true
> **else**
>   CT.Allocated := false
>   *Transition_Rule*
> **endif**

The new object and *type* are now in the typing relation *TR*. We do not update *TR* explicitly, since it is not used in the dynamic semantics.

See the first paragraph in section 4.4 for a good example of ground types whose behavior describes the modes copy and create. The reader may wish to examine it now.

3.3 Statements, Expressions, Variables and Designators

We informally describe the dynamic behavior of the subtrees whose roots are labeled with *Statement, Expression, VarObject, VarParamObject, ValParamObject or Designator*. By the behavior of a subtree we mean the actions that take place from the state when its initial node gets control to the state when its terminal node releases the control. In addition we describe how the information computed while the subtree had the control can be accessed afterwards. The functions used for this are only defined at run time and are set to *undef* in the initial state (e.g. Value, Assoc).

Statements A statement subtree typically executes its work and passes control to the task indicated by the NT field of its terminal node, with the following two exceptions. In the case of a runtime error, control is passed to *Trap*, and in the case of a jump instruction, control might be passed to the end of the respective outer control structure.

Expressions An expression subtree is evaluated, and the calculated result is assigned to the field *VALUE* of its terminal node. If no runtime errors occur, control is passed to the task indicated by the *NT* field of its terminal node.

If the calculated result is not compatible with the type indicated by the *Type* field of its terminal node (e.g. overflow) a runtime error is triggered. Evaluation of Oberon expressions may incur side effects.

In our first formulation we do not consider recursion. In order to reuse the transition rules when we introduce recursion, we define VALUE as a macro, whose following definition will be redefined later on. The codomain of *VALUE* is and remains *OberonValue*.

$$VALUE(e) \triangleq Value(e) \tag{1}$$

Expressions in Oberon are typed. An expression's type can be calculated statically from the types of its subexpressions, and the result of this calculation is assumed to be stored in a field *Type* of the terminal node of the expression subtree.

$$Type : Node \rightarrow Type$$

Variables Subtrees generated by the nonterminals *VarObject*, *VarParamObject* and *ValParamObject* consist of one node and are called variables. Since variables are as well identifiers, the type field is set according to the definition in section 2.2.

A variable is related with the store by its *associated object*. The macro *ASSOC* is used to store this object and is defined as follows:

$$ASSOC(v) \triangleq Assoc(v) \tag{2}$$

Again a macro is used, because its definition will be refined later in order to handle recursion. The codomain of *ASSOC* is and remains *Object*.

A variable either creates or identifies an object of its type in the store. This object is initialized if necessary, and then is assigned to the *ASSOC* field; control is always passed to the task stored in the *NT* field [5].

Designators As expressions, designators are typed. This type is calculated statically and assigned to the *Type*-field of the terminal node of a designator. A designator references an object in the store. It identifies this object and assigns it to the *ASSOC* field of its terminal node. If its type is unstructured, the *Value* of the associated object is assigned to the *VALUE* field of the terminal node. This work is done by the macro **SET_TO**(*object*), which sets the associated object of the current task to *object*. If the type of the current task is unstructured, it also sets the value of the current task to the value of the object. Control is then passed to the next task:

SET_TO(*object*) $\triangleq$

CT.ASSOC := *object*
if UnstructuredType(CT.Type) **then**
  CT.VALUE := *object*.Value
endif
CT := CT.NT

If the type of the associated object of a designator is not compatible with the type of the designator, an exception is triggered and control is passed to *Trap*. Otherwise the control is passed to the *NT* of the terminal node either implicitly by the **SET_TO**-macro or explicitly. Since identification of the associated object may include evaluation of expressions, designators may incur side-effects.

4 Refinements

In the previous sections we gave most of the structures of the initial state on which we model the dynamic semantics and gave informal descriptions of the dynamics. In this section we refine that information, giving the fine grained structure and the exact transition rules for each production in the grammar.

[5] Because we assume an unlimited store, no runtime error can occur.

In subsection 4.1 we consider the production rules *Module* and *ProcDeclaration* which are related to the main quanta of action in Oberon, namely module initializations and procedure calls. The other *language constructs* are grouped into statements, expressions, types, designators and variables. For each of these there is a subsection in which we describe the dynamic behavior and the fine grained structure of the generated subtrees. The informal description of the dynamic behavior given in section 3.2 for types and in section 3.3 for the other constructs is sufficient to understand the dynamic behavior of one specific subtree: it will be defined in terms of the direct descendants of its root, which are either leaves or subtrees described in sections 3.2 and 3.3. The only static structure we use from those subtrees is the initial and terminal nodes; the informal information about their behavior is sufficient for intuition.

We abstract away from recursion until the last subsection. In that subsection we refine the introduced macros in such a way that the transition rules remain valid for the new model including recursion.

The figures in this section are all related to production rules n ::= E. They consist typically of a box representing a derivation tree generated by n and a graph within that box. Boxes in this graph represent subtrees generated by nonterminals in E, and circles represent leaves representing terminals in E. The branches are not shown because they follow from the rule and our definition of the compact derivation trees.

To represent the fine grained control and data flow with these graphs, we use the following conventions: arrows entering and exiting from top or bottom of a box in the graph are considered as data flow and are therefore bound to the terminal node of that box. Control flow arrows, mainly labeled with NT, enter and exit from left and right of the box. Entering control flow is bound to the initial node and exiting control flow to the terminal node of the box.

To define the initial state of a program we build up the structure as defined for the single production rules while we build up the derivation tree. Figure 2 illustrates this process for different abstractions of the subtree c := a + b. First the statement box is shown with two subtrees, represented by boxes. Then one of the boxes in the graph, that of a + b, is further refined. The process is finished when there are no more boxes, which is equivalent to the situation when there are no more nonterminals.

For each construct we give first the corresponding EBNF rule and then the transition rule that deals with this construct. The corresponding part of the initial state is typically given as a graph and the transition rule is shortly explained. The explanations will be liberal and take advantage of the intuitive names.

4.1 Modules and Procedure Declarations

Program	::=	{Module}

The program is a list of *modules*. In order to simplify the situation we will initialize the first module in order to execute the program. A more general view, implemented in the Oberon operating system [38] is that the user nondeterministically executes parameterless procedures that change the current state of the memory(as all user do from the system's point of view). The initialization of modules is then only used to provide an initial state of the memory as a base for these parameterless procedures. Note that, for instance, mouse moving is such a procedure as well and that changing the state of the monitor is often reflected in the memory and can therefore influence the behavior of future calls. We could give an abstract model of this situation in order to simulate the real situation.

Module	::=	MODULE ModObject ";" [ImportList] DclSequence [BEGIN StatementSequence] END ident "."
ModObject	=	CodeObject

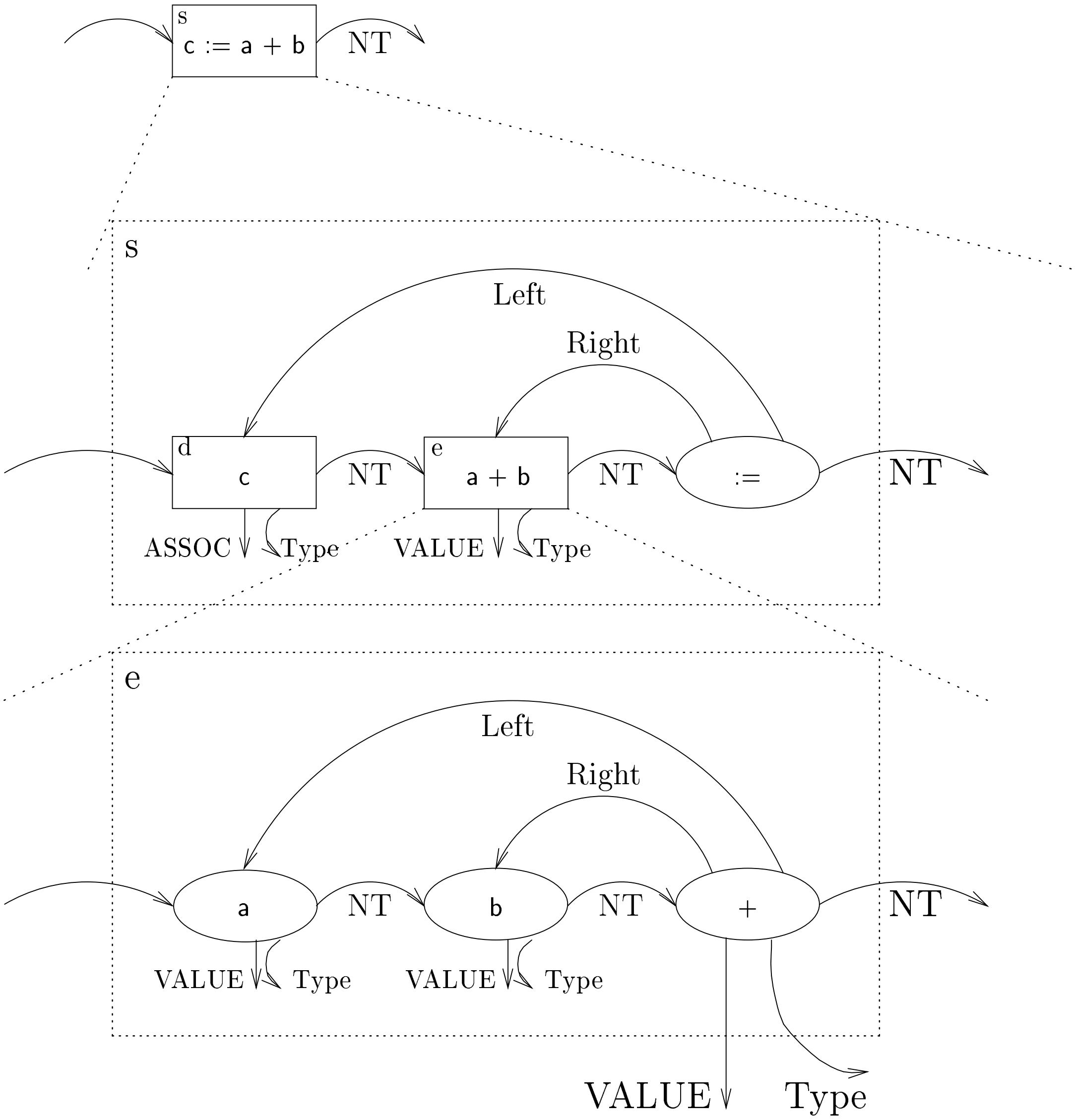

Figure 2: Different refinement stages of c := a + b

The control flow of the initialization of a module can be easily explained in terms of a *NextTask*-linked list: the first element of the list is the module object, followed by the import objects of the import list and then the var objects of the declaration sequence. The order of the import objects among them is not guaranteed. The order of the var objects is not guaranteed either but has no influence on the module initialization.

The next task of the last element l of the list is the initial node of the statement sequence. The next task of the terminal node of the statement sequence is the module object. If there is no statement sequence, the next task of l is the module object.

The module object is therefore the anchor of a *NextTask*-linked ring list. To avoid multiple execution of this ring, a dynamic unary function *State* is used. As usual, this function is by default defined to deliver *undef.* The pseudo-code executed when the module object gets the control will set its state to *initialized* and passes the control on the ring. If the control is back, the value of state will already be *initialized* and the control is no more passed to the ring. At this point we say the module is initialized.

The purpose of a second dynamic field of the module object becomes clear only after we explained the import mechanism in our model.

ImportList	::=	IMPORT Import {"," Import } ";"
Import	::=	ImportObject [":" ExternalName]
ImportObject	=	CodeObject
ExternalName	=	ident

The semantics of the import list are that each imported module must be initialized. The import objects can be seen as procedure calls to the initialization control flow of the respective module. We need a *Return*-field to store the node to which control is returned after the module is initialized. If the current task is a module object then the following happens. If the state is initialized then control is immediately passed to the node stored in the return field; otherwise the current state is set to initialized and control is passed to the next task.

```
if ModObject(CT) then
    if CT.State = initialized then
        CT := CT.Return
    else
        CT.State := initialized
        CT := CT.NT
    endif
endif
```

The *Decl*-field of the import object points to the module object having the same micro syntax, as described in section 2.2. If the current task is an import object the following rule is executed.

```
if ImportObject(CT) then
    CT.Decl.Return := CT.NT
    CT := CT.Decl
endif
```

The return field of the referenced module-object is set to the next task, and control is passed to the referenced module-object. From the semantics of module-objects we know that they pass control back to their return address either directly or after initialization.

Procedures For convenience we repeat the *ProcDeclaration*-rule given earlier with some non-terminals replaced with their definitions.

ProcDeclaration	::=	PROCEDURE ProcObject
		["(" [FPSection {";" FPSection }] ")" [":" ResultTypeId]] ";"
		DclSequence
		[BEGIN StatementSequence] ProcedureEnd ident
ResultTypeId	=	TypeIdent
ProcedureEnd	=	END

The control flow of the execution of a procedure is again an NT-list: the beginning of the list consists of all variable- and value-parameter objects in the *FPSection*-subtrees. Their order in the list is not determined, but we define a static function

$$Position : VarParamObject \cup ValParamObject \to Nat$$

to hold their *syntactical position* from left to right, starting with one. This position is used to actualize the formal parameters.

The NT-list of parameters is continued with the *VarObject*-nodes in the *DclSequence*-subtree and the statements in the *StatementSequence*. The NT-field of the last node of the list is set to the procedure end node.

The additional non terminal *ProcedureEnd* includes the END-nodes of procedure declarations in a universe *ProcedureEnd*. We can use this information to give them specific dynamic behavior. The purpose of procedure-end nodes is to pass control back to the node that called the procedure. We will model procedure calls together with recursion in section 4.7.

4.2 Statements

StatementSequence	::=	Statement {";"Statement}

The statements in a statement sequence are sequentially linked through the next task function in their syntactical order. The initial node of the statement sequence is the initial node of its first statement. The NT-field of the terminal node of one statement is set to the initial node of the next statement in the sequence. The terminal node of the sequence is the terminal node of the last statement in the sequence. The different statements that can be sequenced are:

Statement	::=	Assignment | NewStatement | ProcedureCall |
		IfStatement | CaseStatement | WhileStatement |
		RepeatStatement | LoopStatement | EXIT |
		WithStatement | ReturnStatement | NullStatement
NullStatement	=	

Assignment

Assignment	::=	Designator ":=" Expression

The universe := containing the nodes generated by terminal ":=" may be syntactically misleading. Therefore we write ":=" rather than := to denote it in transition rules. An assignment of the form L := R has data and control flow functions defined on its derivation tree as pictured in figure 3. From the picture we see that designator L has control first and then the expression R. Again this order is not guaranteed. Finally control is passed to the ":="-node. The initial node is the initial node of the designator, and the terminal node is the ":="-node.

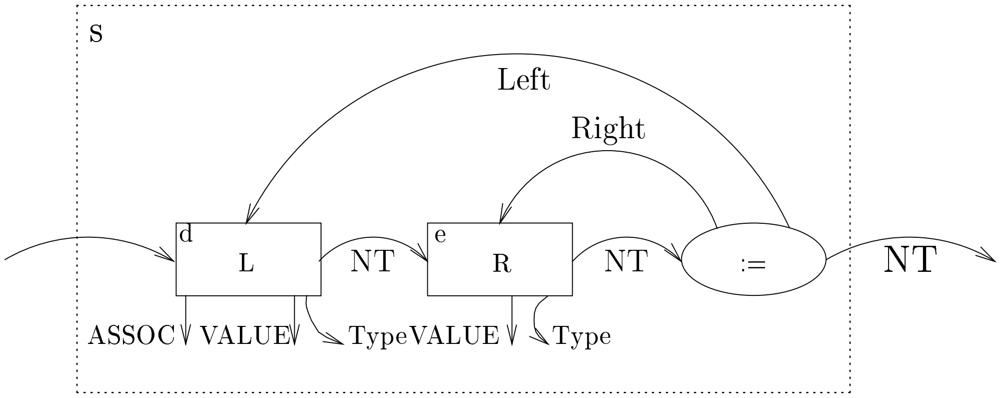

Figure 3: An assignment

```
if ":="(CT) then
    if Unstructured(CT.Left.Type) then
        CT.Left.ASSOC.Value := CT.Right.VALUE
        CT:= CT.NT
    else (* CT.Right must be designator *)
        COPY(CT.Left.Type, CT.Right.ASSOC, CT.Left.ASSOC)
    endif
endif
```

Let the current task be a ":="-node. If the type of the right expression is unstructured, then the assignment task simply sets the *Value* field of the associated object *CT.Left.ASSOC* to *CT.Right.VALUE* and passes control to *CT.NT*. Otherwise the expression *CT.Right* is a designator and therefore has an *ASSOC* field as well. In this case the **COPY** macro is used to assign the values of *CT.Right.ASSOC* to the appropriate fields of *CT.Left.ASSOC*. The **COPY** macro passes control implicitly to *CT.NT*.

Memory allocation

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

In Oberon memory is allocated using the predefined procedure NEW which takes a designator as an argument. The structure of statement NEW(P) is pictured in figure 4. The argument P

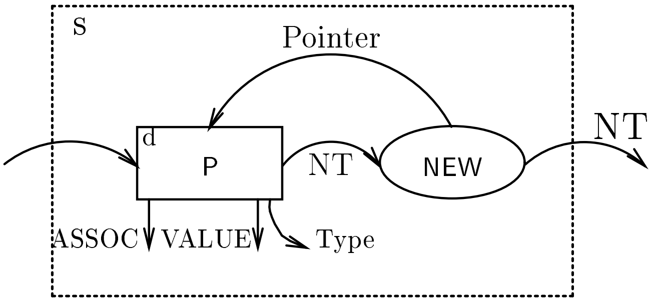

Figure 4: A new statement

receives control before the new-task. The figure defines that the NEW-task is linked with its argument via the static function

$$Pointer : New \rightarrow Designator$$

The type of the object to be allocated by NEW-node n is statically defined and equals the reference type of *n.Pointer.Type*.

The task of a NEW-node n is to allocate a new object and to assign it to the *Value*-field of the associated object of the pointer *n.Pointer*.

```
if NEW(CT) then
    CREATE_A CT.Type THEN
        CT.Pointer.ASSOC.Value := CT.Type.Dest
        CT := CT.NT
endif
```

After the new object is created, the NEW-node assigns it to the *Value*-field of the associated object of the pointer-designator and passes control on to the next task.

Procedure calls

ProcedureCall	=	Designator

The new procedure is a special case of procedure call. We will treat the general case together with recursion in section 4.7.

Control statements

Oberon has if-, while-, repeat-, case- and loop-statements to manipulate the control flow. We restrict ourselves here to the first three. The treatment of case and loop work in the same way.

IfStatement	::=	IF Condition THEN StatementSequence {ELSIF Condition THEN StatementSequence} [ELSE StatementSequence] EndIf
EndIf	=	END
WhileStatement	::=	WHILE Condition DO StatementSequence END
RepeatStatement	::=	REPEAT StatementSequence UNTIL Condition END

We group the nodes representing terminal symbols THEN, DO and UNTIL in a universe *Test*. A test has a static field

$$Condition : Test \rightarrow Expression$$

which points to a (Boolean) expression. A test has two possible successors in the control flow. One is accessed by the static function *TrueTask* and the other by the static function *FalseTask*:

$$TrueTask : Test \rightarrow Node \qquad FalseTask : Test \rightarrow Node$$

If the value of the condition is *true*, the control is passed to the node in the field *TrueTask* otherwise control is passed to the node in the field *FalseTask*.

```
if Test(CT) then
    if CT.Condition.VALUE then
        CT := CT.TrueTask
    else
        CT := CT.FalseTask
    endif
endif
```

The control data flow functions of an if-statement of the form IF E THEN S1 ELSE S2 END are depicted in figure 5. The elsif structures can be translated in nested it-then-else structures. The only subtle point is that there are several terminal nodes, e. g. S1 and S2 in figure 5. For an informal treatment it is sufficient to say that all these nodes play the rôle of the terminal node, but for execution we have to use the *EndIf*-node as target of the many outgoing *NT*-arrows. This node passes control to its next task, and is defined to be the terminal node of the if-then-else construct. We need the following part of the transition rule:

if EndIf(CT) **then**
    CT:= CT.NT
endif

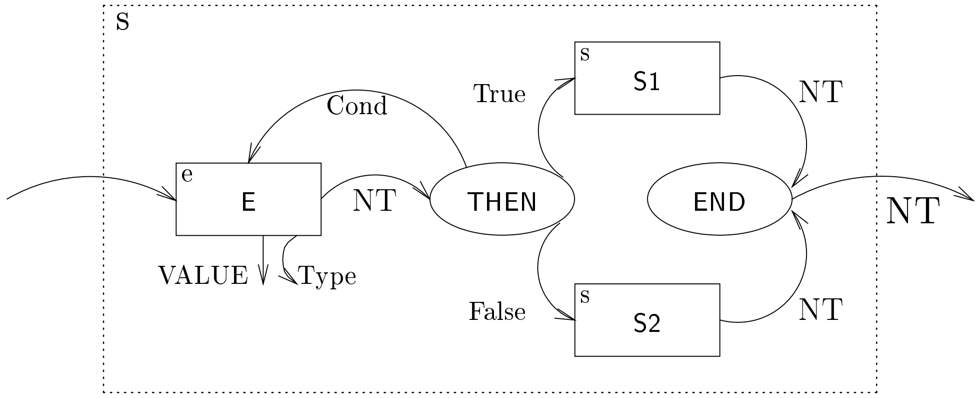

Figure 5: A simple if-statement

A while-statement of the form WHILE E DO S END is pictured in figure 6 and a repeat-statement of the form REPEAT S UNTIL E END is described in figure 7.

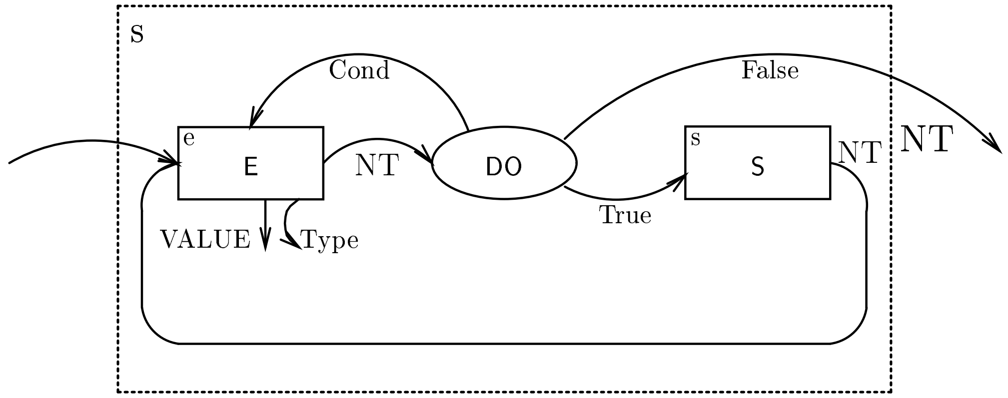

Figure 6: A while-statement

4.3 Expressions

As explained in section 3, an expression evaluates itself and assigns the result r to the *VALUE* field. We say the expression evaluates to r. In this subsection we start by explaining non-composed expressions and then describe composed expressions.

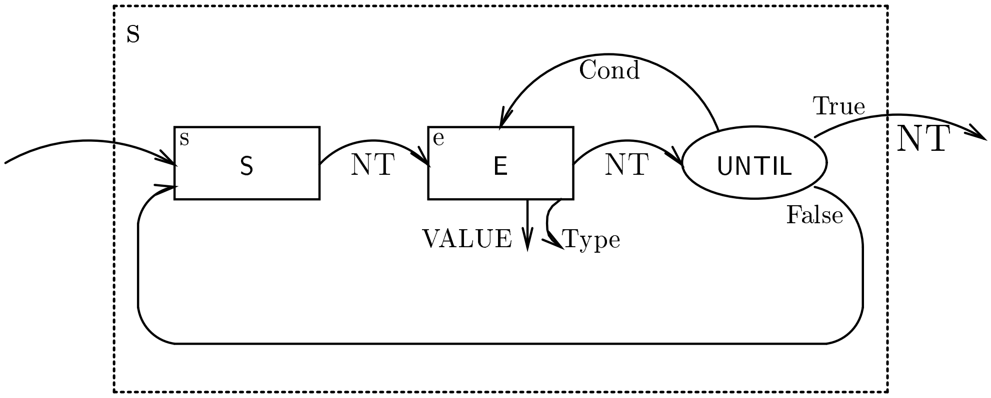

Figure 7: A repeat-statement

The main parts of the expression EBNF-rules were already given as example in section 2.1. As mentioned not all choices for *Factor* were given there. The complete version is:

Factor	=	ExprInParentheses | NegatedFactor | Set | Designator | intnumber | realnumber | character | string | TRUE | FALSE | NIL

Designators as expressions

If a designator occurs as a factor in an expression its type is unstructured. This means that the field *VALUE* was set after the designator had control, so it can be used like a normal expression. The only exception to this rule is when a designator occurs as an expression within an assignment statement (cf. 4.2). For the exact behavior of designators we refer to section 4.5. Here it suffices to know that they behave like normal expressions.

Constants

$$\text{Constant} = \text{intnumber} \cup \text{realnumber} \cup \text{character} \cup \text{string} \cup \{\mathsf{TRUE},\mathsf{FALSE}, \mathsf{NIL}\}$$

Constants evaluate always to the same result, namely to the mathematical object corresponding to their micro syntax. Identifiers referencing constant declarations are replaced by the right-hand side of the corresponding declaration. Composed expressions consisting only of constants are folded to one constant before execution of the program. The type of a mixed expression (e.g. designator and constant) is determined by the type of the non-constant expression. We skip the exact treatment here and use a field *ConstVal* to access the mathematical object corresponding to the micro syntax or the folded composed constant. Of course this does not have to be done at run-time.

We will not go in the details of which numbers are represented by which Oberon number format. The mathematical objects corresponding to the Oberon constants TRUE, FALSE and NIL are the elements *true*, *false* and *Nil*.

```
if Constant(CT) then
    VALUE(CT) := CT.ConstVal
    CT := CT.NT
endif
```

Function calls

The description of functions is deferred to 4.7.

Composed expressions

The context free grammar G_O transforms expressions to a term of unary and binary operations. The control and data flow for the unary expressions *SignedTerm* and *NegatedFactor* is pictured in figure 8. The initial node of such a unary expression is the initial node of the argument expression and the terminal node is the operation symbol.

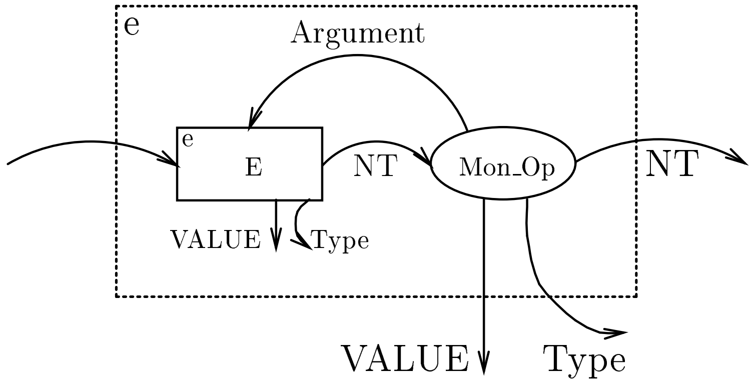

Figure 8: Expression with unary operation symbol Mon_Op

The binary Boolean expressions are different from the other cases and are explained separately. For all the other cases the two arguments are evaluated in an arbitrary order. Thus we can not guarantee the order in which arguments are evaluated. To model this situation we would have to introduce an external function that determines the evaluation order each time a binary expression is evaluated. In order to give a more readable specification, we model only the case of left to right evaluation and stipulate that this order is not guaranteed. The simplified situation is given in figure 9. The Type of Bin_Op can be determined from the types of L and R. The initial node for this fixed evaluation order is the initial node of the left expression, and the terminal node is Bin_Op.
A higher order function

$$Apply : Node \rightarrow (OberonValue \times OberonValue \rightarrow OberonValue)$$

maps nodes labeled with terminals of the set {=, #, <, <=, >, >=, +, -, *, /} to the usual mathematical binary meaning of these terminals, nodes labeled with DIV to the composition of the absolute function and the division function, and nodes labeled with MOD to the usual modulo function. The functions are as well defined for set arguments, but we do not specify the details here.

The static semantics of Oberon guarantees that this mathematical semantics is sufficient in the case of relations and in the case of the modulo operation.

```
if RelOp(CT) or MOD(CT) then
    CT.VALUE := Apply(CT)(CT.Left.VALUE, CT.Right.VALUE)
    CT := CT.NT
endif
```

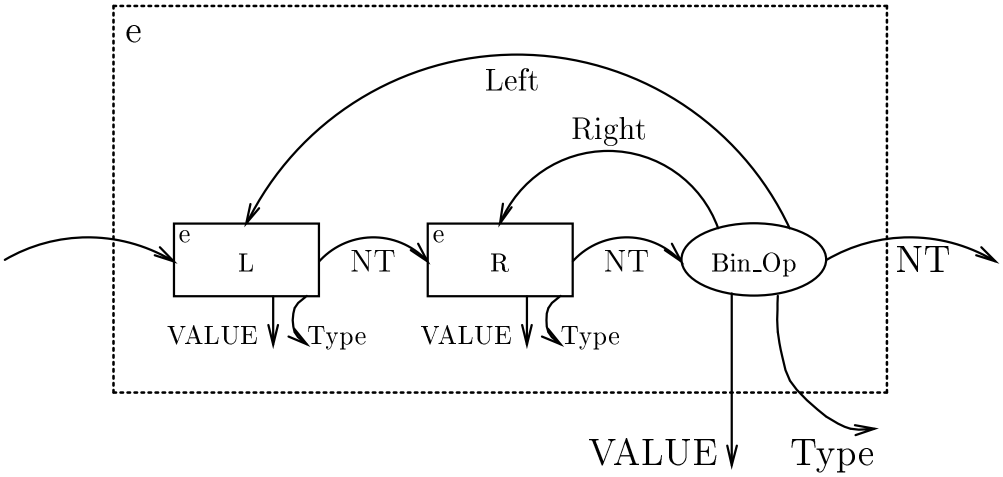

Figure 9: Expression with binary operation symbol Bin_Op

Run-time errors of numeric expressions

Numeric expressions have a semantics which is different from standard mathematics. It can neither be guaranteed that the result of a numeric computation in Oberon is defined (e.g. division by zero), nor that the resulting value is compatible with the type of the expression. In such cases execution is aborted and control is passed to the node *Trap*.

The following macro CHECK is introduced to determine if the result yielded by the evaluation of an expression lies within the range of compatible values of a given numeric type. It takes as arguments an *OberonValue* term value and a ground type and generates a transition rule. The definition of CHECK(value, type) is:

> **if** value $\leq$ Max(type) **and** value $\geq$ Min(type) **then**
>   true
> **else**
>   false
> **endif**

If it occurs in a transition rule it will be replaced by this definition, where the parameter value and type are replaced by the actual arguments of CHECK. The functions Max and Min denote the largest, respectively the smallest member of the universes representing the numeric types. Again we have to consider the case of of a set value. Without giving the details, we define that CHECK($\cdot$, SET) yields always true: set operations are free from run-time errors.

> **if** Sign(CT) **then**
>   **if** CHECK(Apply(CT)(0, CT.Argument.VALUE), CT.Type)) **then**
>   VALUE(CT) := Apply(CT)(0, CT.Argument.VALUE)
>   **else**
>   CT := Trap
>   **endif**
> **endif**

This run-time check seems superfluous, but it is for example not guaranteed that (0 - Max(SHORTINT) $\geq$ Min(SHORTINT).

```
if not Sign(CT) and ("+"(CT) or "-"(CT) or "*"(CT)) then
    if CHECK(Apply(CT)(CT.Left.VALUE, CT.Right.VALUE), CT.Type) then
        VALUE(CT) := Apply(CT)(CT.Left.VALUE, CT.Right.VALUE)
        CT := CT.NT
    else
        CT := Trap
    endif
endif
```

The result of the usual mathematical calculation is tested to suit in the bounded range of numeric types in Oberon. If this range is exceeded a run-time error is raised. Although this property is highly desirable, it is rarely implemented in hardware and therefore poses efficiency problems in software implementations. Most implementations of Oberon do not implement them and are thus incorrect in a strict sense.

```
if "/"(CT) or DIV(CT) then
    if CT.Right.VALUE ≠ 0 and
            CHECK(Apply(CT)(CT.Left.VALUE, CT.Right.VALUE), CT.Type) then
        VALUE(CT) := Apply(CT)(CT.Left.VALUE, CT.Right.VALUE)
        CT := CT.NT
    else
        CT := Trap
    endif
endif
```

When division is involved, we have to test in addition whether the right argument is null. Unlike the mathematical semantics where division by zero is not defined, we enter the defined state *Trap*.

Lazy evaluation of Boolean expressions

Oberon features conditional or lazy evaluation of Boolean connectives. This allows a result of an expression to be well defined even if one of the operands is not. The semantics for lazy evaluation of the connectives:

p1 OR p2	$\Leftrightarrow$	if p1 then TRUE else p2
p1 & p2	$\Leftrightarrow$	if p1 then p2 else FALSE

where p1 and p2 are Boolean expressions. As a consequence of this definition the operators are not commutative.

We see from figure 10, that control is passed to the left expression and then directly to *CondOp*, which can be & or OR. Depending on the value of the left expression, *CondOp* passes control to the initial leaf of the right expression or directly to the next task. To remember whether the right expression has already been evaluated, the field *State* of the *CondOp* is used. The initial node of the Boolean expression is the initial node of the left expression, and the terminal node is the *CondOp*-node.

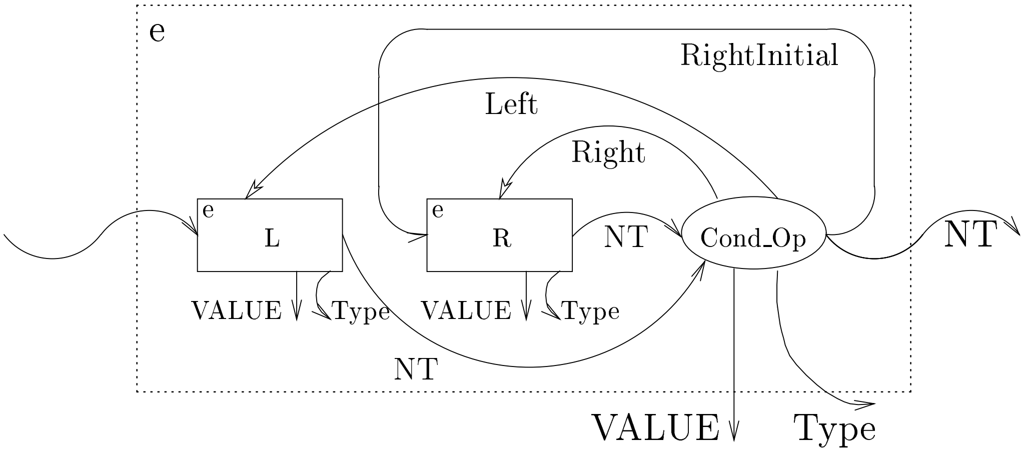

Figure 10: Boolean and-, or-expressions

if "&"(CT) **then**
    if CT.Left.VALUE **then**
        if CT.State = undef **then**
            CT.State := right_visited
            CT := CT.RightInitial
        else
            CT.VALUE := CT.Right.VALUE
            CT.State := undef
            CT := CT.NT
        endif
    else
        CT.VALUE := false
        CT := CT.NT
    endif
endif

If the value of the left expression is *false*, the and-expression is guaranteed to evaluate to *false* and control is passed to the next task of the "&"-node. If the left value is *true* the right expression has to be evaluated in order to determine the value of the and-expression: the state is set from *undef* to *right_visited* and control is passed to the right expression. If the control comes back the state is still *right_visited*. To complete its task the node sets the state back to *undef*, updates its *VALUE*-field with the value of the right expression and passes control to the next task.

```
if OR(CT) then
    if ¬ (CT.Left.VALUE) then
        if CT.State = undef then
            CT.State := right_visited
            CT := CT.RightInitial
        else
            CT.VALUE := CT.Right.VALUE
            CT.State := undef
            CT := CT.NT
        endif
    else
        CT.VALUE := true
        CT := CT.NT
    endif
endif
```

The dynamics of an *OR*-node are similar to those of an "&"-node. The difference is that the result is determined to be *true* if the left expression is *true*. If not the right expression has to be evaluated and its result will be the result of the or-expression.

NegatedFactor	::=	"∼" Factor

The transition rule for logical not is given for completeness:

```
if "∼"(CT) then
    CT.VALUE := ¬(CT.Argument.VALUE)
    CT := CT.NT
endif
```

4.4 Types

Type	=	TypeIdent | GroundType | PointerType | ProcedureType | RecordType | ArrayType
TypeIdent	=	Designator

The eight ground types SHORTINT, INTEGER, LONGINT, REAL, LONGINT, BOOLEAN, SET, and CHAR form a well suited interface to reflect the properties of the underlying computing machine while abstracting from a concrete implementation. The dynamic behavior of all ground types is the same. We use each occurrence of a *GroundType*-node as an active task that reflects this behavior.

```
if GroundType(CT) then
    if Mode = copy then
        CT.Dest.Value := CT.Src.Value
    else (* Mode = create *)
        extend Location with o
            CT.Dest := o
        endextend
    endif
    CT := CT.Return
endif
```

This transition rule exposes the tasks of a type in a nice way. It gives a simple example of what we wanted to express in section 3.2. In copy mode, the value of the destination location is set to the value of the source location, and control is passed to the node in the *Return*-field. In create mode, a new location is created and assigned to the *Dest*-field. Control is passed to the return "address".

PointerType	::=	POINTER TO ReferencedType
ReferencedType	=	Type

One difference with ground types is that objects of pointer types must be initialized by setting their value to *Nil*. As all pointer types have the same dynamic behavior in our model, we can introduce a single type for all of them or use the POINTER-node as type. For simplicity we chose the second solution. The only additional static function we need is

$$\mathit{ReferencedType}\colon \mathsf{POINTER} \to (\mathsf{RECORD} \cup \mathsf{ARRAY})$$

which maps the POINTER-node to its referenced type (e.g. the type of the *ReferencedType*-node.

The predefined object *Nil* is defined to be compatible with every record or array type [6]. We define *(Nil As T) = Nil* for every record type *T*. See the paragraph about record types for more details.

```
if POINTER(CT) then
    if Mode = copy then
        CT.Dest.Value :=   CT.Src.Value As CT.ReferencedType
    else (* Mode = create *)
        extend Location with o
            CT.Dest := o
            o.Value := Nil
        endextend
    endif
    CT := CT.Return
endif
```

The other difference comes up in copy mode: the source has to be converted into its correct instance through *As*, before it can be assigned to the destination.

ProcedureType	::=	PROCEDURE
		["("[ProcTypeParam {"," ProcTypeParam}]")"
		[":" ResultTypeId]]
ProcTypeParam	::=	[VAR] VirtualObject FormalType
VirtualObject	=	

The procedure type behaves like a ground type, but must be initialized with *Nil* on creation. It is first of all used for static type checking.

```
if PROCEDURE(CT) then
    if Mode = copy then
        CT.Dest.Value :=   CT.Src.Value
    else (* Mode = create *)
        extend Location with o
            CT.Dest := o
            o.Value := Nil
        endextend
    endif
    CT := CT.Return
endif
```

[6] In a private discussion N. Wirth agreed to this and decided to incorporate this definition into the standard language definition. His Oberon compiler does not behave like this (e.g. type guards with pointers referencing Nil do not terminate there)

In the following the structured type constructors of Oberon are explained. Their generality is the reason why the solution with the types as active objects has been chosen. Simpler solutions like higher order functions do not scope to the complexity of Oberon type. The active object solution on the other hand would be suited to model even more general type systems.

Record types

RecordType	::=	RECORD ["("BaseTypeId")"] [FieldDeclaration] {";" [FieldDeclaration]} RecordEnd
BaseTypeId	=	TypeIdent
RecordEnd	=	END
FieldDeclaration	::=	FieldObject { "," FieldObject } ":" Type
FieldObject	=	ident

A record type definition consists of a possible reference to a base type and a list of field declarations. These declarations define *fields* of the record type. Visible fields of the base type are inherited and thus as well fields of the record type. If the declaration of the base type is in the same module, all its fields are visible. If it is declared in an imported module, only the exported fields, namely those marked by an asterix are visible. Invisible field names of the base type may be reused as field names of the record type.

Invisible names are necessary for distributed development of software but have consequences on our model. These consequences affect the dynamic semantics as well as the static semantics. It makes it impossible to model the subtype mechanism of record types as simple extensions of the mapping from field names to fields. A concrete example is given in appendix A.

We will now describe the dynamics of a record type, i.e. how this complicated record model is built up in *create* mode and how the information of one record is copied to another in *copy* mode.

The control and data flow structures which are needed for this task are illustrated in figure 11. The static field *BaseType* of the RECORD-node is set to the type of the *BaseTypeId.* If there is no base type, then this field is set to *undef.* The control flow is reflected by a *NT*-list beginning with the RECORD-node containing all field objects and ending with the *RecordEnd*-node.

All field objects and the *RecordEnd*-node are linked with the RECORD-node through the static function *Parent*: *FieldObject* $\cup$ *RecordEnd* $\rightarrow$ RECORD. The type of field objects is defined as the type given in the field declaration.

Create mode Record allocation is a recursive process. For record types that are not extensions the following happens: first a record x must be allocated, then for each field of x an object of appropriate type (possibly a record itself) must be allocated and linked to x. The linkage is done by assigning this new object to *Field(x, name)*, where *name* is the name of the field.

If the record is an extension, first a base record y is created and then the extension x is allocated and inherits all fields of y. This inheritance is done by a higher-order update of the curried version of function *Field*: *Field(y)* is assigned to *Field(x)* [7]. Then the objects for the new fields are allocated and linked. If the field name of a new field is the same as that of an (invisible) inherited field, the link to the inherited field is overwritten.

The two objects x and y are linked by the infix function

$$As : Record \times RecordType \rightarrow Record$$

(e.g. if the type of x is X and the type of y is Y then *(x As Y)* is set to y and *(y As X)* is set to y). Maintaining the reflexive, transitive closure of these links for more than two record types in a subtype relation is a task for the *RecordEnd* node.

[7] This higher-order update can be statically translated to ordinary updates.

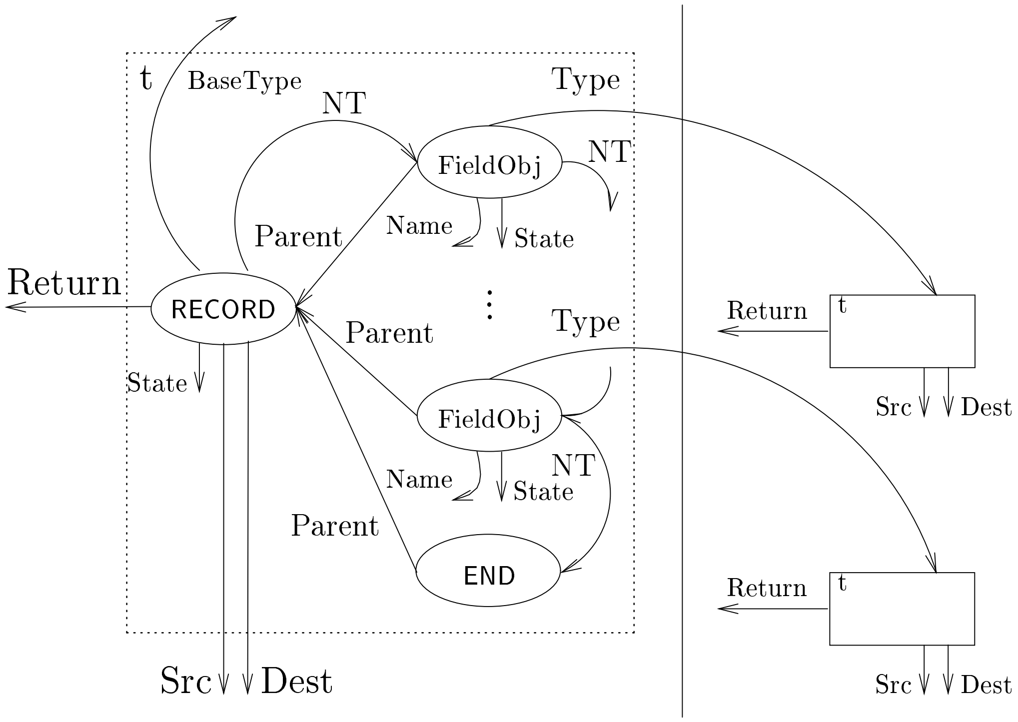

Figure 11: Control and data flow structure of a record type

if RECORD(CT) **and** Mode = create **and** CT.BaseType = undef **then**
  extend Record **with** o
    CT.Dest := o
    CT := CT.NT
  endextend
endif

This transition rule is executed in create mode, if the current task is a record type that is not an extension of another. The current task extends the universe *Record* with a new element and assigns this element to the *Dest* field and passes control to the first field object. The new record is thus accessible for all field objects and the RecordEnd node as the *Dest* field of their parent.

if RECORD(CT) **and** Mode = create **and** CT.BaseType ≠ undef **then**
  CREATE_A CT.BaseType **THEN**
    extend Record **with** o
      Field(o) := Field(CT.BaseType.Dest)
      CT.Dest := o
      (o As CT.BaseType) := CT.BaseType.Dest
      (CT.BaseType.Dest As CT) := o
      (o As CT) := o
    endextend
    CT := CT.NT
endif

The preceding rule is executed if the record type is an extension. The record type creates an object of its base type. We know that this new object is accessible as the *Dest*-field of the base type. Then it creates another new object and assigns it to its own *Dest*-field. The fields of the base type object are inherited by this object through the higher-order update *Field(o) := Field(CT.BaseType.Dest)*, and the link between the base type object and the new extension object is established in both directions and reflexive on the new object. The control is then passed to the next task, which is typically the first field object [8].

if FieldObject(CT) **and** Mode = create **then**
  CREATE_A CT.Type **THEN**
    Field(CT.Parent.Dest, CT.Name) := CT.Type.Dest
    CT := CT.NT
endif

A field object in create mode creates an object of its type, links the newly created object to its parent by assigning it to the function *Field* at the point (*CT.Parent.Dest, CT.Name*), and passes control to the next task in the field-declaration list.

if RecordEnd(CT) **and** Mode = create **then**
  As := TransAs
  CT := CT.Parent.Return
endif

The field object list is terminated by a *RecordEnd* node. Its purpose is to maintain the transitive closure of the *As*-function by updating it with *TransAs*. The definition of this higher-order update can be statically translated into ordinary updates. In order to define *TransAs* we introduce a ternary relation

$$\mathit{TransAsClosure} : \mathit{Record} \times \mathit{RECORD} \times \mathit{Record} \rightarrow \mathit{Bool}$$

with the following recursive definition:

$$\begin{aligned} &\mathit{TransAsClosure}(r_1, t, r_2) \iff \\ &\qquad r_1 \; As \; t = r_2 \\ &\quad \vee \quad (\exists R', T' : \mathit{Record}(R') \wedge \mathit{RECORD}(T') : \\ &\qquad\quad R_1 \; As \; T' = R' \quad \wedge \quad \mathit{TransAsClosure}(R', T, R_2)) \end{aligned}$$

For the execution of the specification and for the translation of the higher-order updates, this recursive definition is guaranteed to terminate if the $\vee$ is lazily evaluated and the *Record* and *RecordType* universes are finite. The definition of *TransAs* is now for all r in *Record* and t in RECORD:

$$(r_1 \; TransAs \; t) = \begin{cases} r_2 & : \; \{r_2\} = \{r \mid \mathit{TransAsClosure}(r_1, t, r)\} \\ undef & : \; otherwise \end{cases}$$

The generation of a non recursive sequential EA solution is not trivial, and it might be interesting to allow such fix point constructions.

Copy mode

if RECORD(CT) **and** Mode = copy **then**
  if CT.BaseType = undef **then**
    CT := CT.NT
  else
    COPY(CT.BaseType, (CT.Src As CT.BaseType), (CT.Dest As CT.BaseType))
  endif
endif

[8] Note that the updates are executed in parallel, but we describe them for convenience in a sequential manner.

In copy mode a record type does the following: whenever the record type R has a base type, it copies the values of the base type "version" of the source to the base type "version" of the target. Invisible fields of the base type are copied like this as well. If there is no base type, control is passed directly to the field objects.

```
if FieldObject(CT) and Mode = copy then
    COPY(CT.Type, Field(CT.Parent.Src, CT.Name), Field(CT.Parent.Dest, CT.Name))
endif
```

The task of a field object with name n is to perform a copy action of the n fields of its parent's source and destination. In order to do this it assigns these two objects to source and destination of its own type.

```
if RecordEnd(CT) and Mode = copy then
    CT := CT.Parent.Return
endif
```

The record end in create mode passes control to the return address of the parent's type.

Array types

The type ARRAY a,b,...,z OF T is an abbreviation for ARRAY a OF ARRAY b OF ARRAY ... OF ARRAY z OF T. We translate the abbreviated form to the long one.

ArrayType	::=	ARRAY Length OF ElementType
Length	=	ConstExpression
ElementType	=	Type

The type of the elements of an array is stored in the field *ElementType* of the ARRAY-node.

The following transition rules are specific algorithms to copy or create the elements of an array. The field *Current* is used as help variable in these loop-algorithms which are just doing the usual copy and create actions for all fields of an array.

The fact that open arrays are a feature of Oberon obliges us to redefine the *Length*-field of an ARRAY-node, when recursion is modeled. We define it as the value of the *Length* in the ArrayType production rule for now. In order to reuse the following transition rules, if recursion is modeled, we introduce a macro *LENGTH*. The definition is *LENGTH(a)* $\triangleq$ *Length(a)* and will be refined later. The codomain of *LENGTH* is and will remain *Nat*.

```
if ARRAY(CT) and Mode = copy then
    if CT.Current = undef then
        CT.Current := 0
    else
        if CT.Current < CT.LENGTH then
            CT.Current := CT.Current + 1
            CT.ElementType.Return := CT
            CT.ElementType.Src := Index(CT.Src, CT.Current)
            CT.ElementType.Dest := Index(CT.Dest, CT.Current)
            CT := CT.ElementType
        else
            CT.Current := undef
            CT := CT.Return
        endif
    endif
endif
```

We cannot use the macro **COPY** since control goes back to the type and not to the next task, as assumed by **COPY**.

```
if ARRAY(CT) and Mode = create then
    if CT.Current = undef then
        extend Array with o
            CT.Dest := o
            CT.Current := 0
            CT.ElementType.Return := CT
            CT := CT.ElementType
        endextend
    else
        if CT.Current < (CT.LENGTH-1) then
            Index(CT.Dest, CT.Current) := CT.ElementType.Dest
            CT.Current := CT.Current + 1
            CT.ElementType.Return := CT
            CT := CT.ElementType
        else (* CT.Current = CT.LENGTH-1 *)
            Index(CT.Dest, CT.Current) := CT.ElementType.Dest
            CT.Current := undef
            CT := CT.Return
        endif
    endif
endif
```

The formal types include open arrays. The parameter actualization will update the *LENGTH*-field of an open array with the current length. Therefore the following productions need no additional transition rules.

FormalType	=	OpenArray | TypeIdent
OpenArray	::=	ARRAY OF FormalType

4.5 Designators

Designator	=	ident | Dereferenced | Qualified | Indexed | GuardOrCall
Dereferenced	::=	Designator"↑"
Qualified	::=	Designator "." FieldSelector
FieldSelector	=	ident
Indexed	::=	Designator"["Index"]"
Index	=	Expression
GuardOrCall	::=	Designator"(" [ExpList] ")"
ExpList	::=	Expression ["," ExpList]

Before focusing on the dynamics of designators we have to tackle two ambiguities in the Oberon syntax.

The first ambiguity concerns the nonterminal *Qualified*. It is not decidable in a context-free manner whether the *Qualified*-term D.F is a *record field* or a reference to a code object of module D. But in section 2.2 we included the *FieldSelector*-nodes referencing to some imported module in *BasicReference*. By defining the initial node of the *Qualified*-node n depending on this, we can lead the control flow directly to external references.

$$Initial(n) = \begin{cases} n.FieldSelector & : \; BasicReference(n) \\ Initial(n.Designator) & : \; \neg BasicReference(n) \end{cases}$$

The above definition of the initial function allows us to treat internal and external references to variables and procedures in a uniform manner:

Direct references to variables or procedures

> **if** BasicReference(CT) **and** VarObject(CT.Decl) **then**
>   **SET_TO**(CT.Decl.ASSOC)
> **endif**

A variable reference sets its *ASSOC*-field to the associated object of its referenced variable. If its type is unstructured, then both *ASSOC*- and *VALUE*-fields are copied. This is done using the **SET_TO** macro. We say in the following only, the designator is set to its associated object.

> **if** BasicReference(CT) **and** ProcObject(CT.Decl) **then**
>   CT.VALUE := CT.Decl
>   CT := CT.NT
> **endif**

A procedure reference evaluates to its referenced procedure. Its associated object is undefined, and it cannot be used as argument for further designator constructions.

The second syntactical ambiguity in Oberon is that it cannot be decided by the context free syntax whether a *GuardOrCall*-term D(P) is a function call or a *type guard* unless we know whether identifier P is a type or not. We will thus make a case distinction in the static semantics in *GuardOrCall*-nodes g satisfying *TypeObject(g.ExpList.Expression.Decl)*, so called *type guards*, and the others, so called *procedure calls*. The former are represented in the paragraph about type guard and the latter are presented in section 4.7.

We can now present the transition rules and the structure for dereferenced designators, record field selectors, indexed designators , dereferenced designators and type guards.

The terminal nodes of the designator constructs are: the uparrow "↑" for dereferenced designators the *FieldSelector* for record field selectors, the second bracket "]" for indexed designators, and the identifier between the "(" and the ")" brackets for type guards. All designator constructs have one designator node which is accessed by the *Argument*-field of the terminal nodes. The initial node of all five constructs is the initial node of their designator argument. Figure 12 shows the control and data flow structure of a typical designator with terminal node *terminal* and argument D. This picture is accurate for all constructs except indexed designators, which have an additional argument.

Dereferenced designators

The type of the "↑"-node is defined to be the referenced type of the (pointer) type of its argument.

> **if** "↑"(CT) **then**
>   **if** CT.Argument.VALUE $\neq$ Nil **then**
>   CT.ASSOC := CT.Argument.VALUE
>   CT := CT.NT
>   **else**
>   CT := Trap
>   **endif**
> **endif**

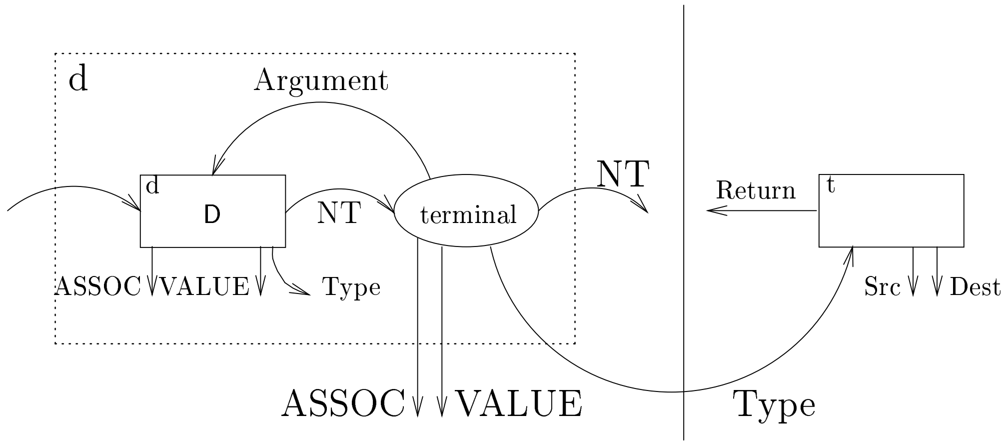

Figure 12: A typical designator construct

The dereferencing "↑" has an argument whose associated object is guaranteed to be of pointer type. Pointer types are unstructured, and the value of the argument is the object the pointer points to. If this object is *Nil*, it results in a run-time error and control is passed to *Trap*. Otherwise the pointed object gets the associated object, and control is passed to the next task. As the object pointed to is of record or array type, we do not have to set the value.

Record field selector

As mentioned above, only field selectors that are not basic references are real record field selectors; the others are external references. The type of a record field selector is defined to be the type of the corresponding field object of its arguments (record) type. The field object corresponding to a field selector is the one having the same name. If the type of the argument is a pointer type, we have to search the field object in the referenced record type.

```
if FieldSelector(CT) and not BasicReference(CT) then
    if RECORD(CT.Argument.Type) then
        SET_TO(Field(CT.Argument.ASSOC, CT.Name))
    endif
endif
```

The normal record field selection applies the *Field* function to the associated object of the argument to get the field object. The second argument of the *Field* function is the name of the field. The value is retrieved if the type is unstructured (by use of **SET_TO**).

```
if FieldSelector(CT) and not BasicReference(CT) then
    if POINTER(CT.Argument.Type) then
        if CT.Argument.VALUE ≠ Nil then
            SET_TO(Field(CT.Argument.VALUE, CT.Name))
        else
            CT := CT.Trap
        endif
    endif
endif
```

If the argument is of pointer type, the record is accessed as the value of the argument. This is called implicit dereferencing. If this value is *Nil*, a run-time error occurs: control is passed to *Trap*.

Indexed designator

Indexed designators look slightly different from the other designator constructs (figure 13). Their second argument is accessed by the *IndexArgument* function. The depicted control flow is not guaranteed. One could just as well evaluate first the index argument and then the designator argument. The type of the "]"-type is defined to be the element type of the type of its argument or, if its argument is of pointer type, the element type of the referenced type of the type of its argument.

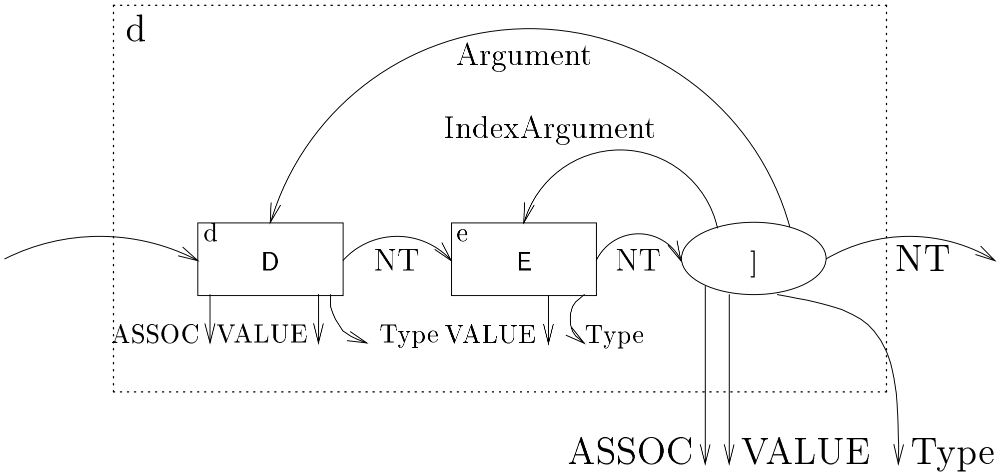

Figure 13: An indexed designator

if "]"(CT) **then**
 if (CT.IndexArgument.VALUE $\geq$ CT.Type.LENGTH) **or**
  (CT.Number.VALUE < 0) **then**
   CT := Trap (* out of bounds run-time error *)
 else
  NORMAL_CASE
  IMPLICIT_DEREFERENCE
 endif
endif

Access to an array's fields is subject to run-time bound checks in Oberon. If the value of the index argument is smaller then zero or greater than or equal to the length of the array, the access is out of bounds and control is passed to *Trap*. Otherwise the block of the two macros *NORMAL_CASE* and *IMPLICIT_DEREFERENCE* is executed.

NORMAL_CASE $\triangleq$
 if ARRAY(CT.Argument.Type) **then**
  SET_TO(Index(CT.Argument.ASSOC, CT.IndexArgument.VALUE))
 endif

In the normal case we have the array directly as the associated object of the argument. We can thus apply the *Index*-function on this object, in order to get the associated object and eventually the value of the "]"-node.

IMPLICIT_DEREFERENCE $\triangleq$
    if POINTER(CT.Argument.Type) **then**
        if CT.Argument.VALUE $\neq$ Nil **then**
            SET_TO(Index(CT.Argument.VALUE, CT.IndexArgument.VALUE))
        else
            CT := Trap
        endif
    endif

The updates in the *IMPLICIT_DEREFERENCE* macro are fired whenever the associated object is a pointer to the array. We can again use the value field to get the array. The only other thing to do in this case is to check whether *Nil* is being dereferenced and to raise a run-time error in that case.

Type guards

The terminal node of a type guard is an identifier whose declaration is a type object. Its type is thus already defined by the general rules in section 2.2. Its task is to check whether the associated object of its argument can be interpreted as being of that type. If this is not possible, the *As* function produces *undef* and control is passed to *Trap*.

    if TypeObject(CT.Decl) **and** RECORD(CT.Type) **then**
        if (CT.Argument.ASSOC As CT.Type) $\neq$ undef **then**
            CT.ASSOC := (CT.Argument.ASSOC As CT.Type)
            CT := CT. NT
        else
            CT := Trap
        endif
    endif

If the guard type is a record type, the associated object is updated with its interpretation as the guard type.

    if TypeObject(CT.Decl) **and** POINTER(CT.Type) **then**
        if (CT.Argument.VALUE As CT.Type.ReferencedType) $\neq$ undef **then**
            CT.ASSOC := CT.Argument.ASSOC
            CT.VALUE := (CT.Argument.VALUE As CT.Type.ReferencedType)
            CT := CT.NT
        else
            CT := Trap
        endif
    endif

If the guard type is a pointer type, the associated object is copied from the argument and the value is updated with its interpretation as the referenced type of the guard type. If the pointer points to *Nil*, nothing special happens. [9]

4.6 Variables and Parameters

The *VarObject* nodes are called *global variables* if they appear in a variable declaration of a module's declaration sequence, and they are called *local variables* if they appear in the declaration

[9]The compiler of N. Wirth produces a run-time error in this case, but he advised us to specify otherwise, in order to make the language cleaner.

sequence of a procedure declaration. Global and local variables have the same dynamic behavior in our model. *VarParamObject* nodes are called *variable parameters*, and *ValParamObject* nodes are called *value parameters*.

We refer to all three types of code objects as *variables* because they are all equivalent for the user of the code object: they have an associated object in the store which is the current “value” of the variable. The dynamic behavior of them is defined in the following.

Global and local variables

```
if VarObject(CT) then
    CREATE_A CT.Type THEN
        CT.ASSOC := CT.Type.Dest
        CT := CT.NT
endif
```

A global or local variable allocates an object of its type and assigns it to its *ASSOC*-field. The only difference between global and local variables is that global variables receive control once, when the module containing them is initialized, while local variables receive control each time the procedure that contains them is called.

Formal parameters

A formal parameter is actualized with an actual parameter if it receives control. The formal parameters are within a procedure and therefore get control only if the procedure is called. The node that called the procedure is stored in the nullary macro *CALLER*. It is defined as a nullary node-valued function *Caller* and will be refined in section 4.7.

The link between calling node and actual parameters is given by a static function

$$ActualParameter : Node \times Nat \to Node$$

which maps the caller and the position *Position* of an actual parameter to the terminal node of this actual parameter.

A **value parameter** allocates an object of its type and copies the value(s) of the corresponding actual parameter in this object. It can be used afterwards like a local variable. The copy mechanism of the type of the value parameter is used to copy whenever the type is not unstructured.

```
if ValParamObject(CT) then
    if CT.State = undef and ARRAY(CT.Type) then
        CT.Type.LENGTH :=
            ActualParameter(CALLER, CT.Position).Type.LENGTH
    endif
    CREATE_A CT.Type THEN
        CT.ASSOC := CT.Type.Dest
        if UnstructuredType(CT.Type) then
            CT.Type.Dest.Value :=
                ActualParameter(CALLER, CT.Position).VALUE
            CT := CT.NT
        else
            COPY(CT.Type,
                    ActualParameter(CALLER, CT.Position).ASSOC,
                    CT.Type.Dest)
        endif
endif
```

The value parameter object uses the **CREATE_A** macro to create an object of its type. In the first intermediate state of the **CREATE_A** macro, characterized by *CT.State = undef*, it copies the length of the type of the actual parameter to its type, if the type is an array type. This is done to actualize the length of open array parameters. After the new object is allocated, it does the following: if it has an unstructured type, then the *VALUE* of its actual parameter is copied to the value of the new object *CT.Type.Dest*. Otherwise the new object is given as the source and the *ASSOC* of the actual parameter as the destination of the *COPY* macro.

A **variable parameter** assigns the associated object of the current actual parameter to its *ASSOC*. Manipulations on this variable are therefore direct manipulations of the corresponding actual parameter.

```
if VarParamObject(CT) then
    CT.ASSOC :=
      ActualParameter(CALLER, CT.Position).ASSOC
    if ARRAY(CT.Type) then
        CT.Type.LENGTH :=
            ActualParameter(CALLER, CT.Position).Type.LENGTH
    endif
    CT := CT.NT
endif
```

Again we have to actualize the length of possibly open array types.

4.7 Procedures and Functions

During execution five kinds of results are produced by the already executed tasks and can be used by the tasks still to be executed:

- The result of the evaluation of an expression E is assigned to VALUE(terminal(E)).
- The allocated or copied associated object of a variable V is stored in ASSOC(V).
- The calculated associated object (and eventually value) of a designator D is stored in ASSOC(terminal(D)) (respectively in VALUE(terminal(D)).
- The actual length of an open array type T is stored in LENGTH(initial(T)).
- The caller of the currently active procedure is stored in CALLER.

If the same task is executed twice (e.g. if it is part of a loop) the results stored by *VALUE*, *ASSOC* and *LENGTH* are overwritten. While, repeat, and loop statements rely on this behavior.

Recursion Procedures (functions) consist of tasks and are named units of execution. This units can be executed by calling them. Such a call is an ordinary statement (respectively expression if a result is returned). As a procedure (function) contains statements, it can contain a call to itself as well. This leads to the situation that the tasks of the itself calling procedure (function) 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 solve this problem we parameterize the functions used in the macros *VALUE*, *ASSOC*, *LENGTH* and *CALLER* with the recursion level. Typically one uses a *stack* for this and calls a section containing informations of one recursion level a *frame*. We abstract from such implementation details and introduce a nullary dynamic function *CurrentRecursionLevel* (or short *CRL*) that points always to the current recursion level.

$$CRL : Nat$$

In the initial state *CRL* is initialized with 0. If a procedure (function) is called, the recursion level is incremented by one and the formal parameters are actualized. After execution the control is given back and the recursion level decremented by one. Thus the information calculated before the call is still available.

We mentioned that we have to parameterize the value, assoc, length and caller function with the recursion level. We therefore introduce four new functions having all an additional first argument for the recursion level:

RecLevelValue: $Nat \times Node \rightarrow OberonValue$
RecLevelAssoc: $Nat \times Node \rightarrow Object$
RecLevelLength: $Nat \times$ ARRAY $\rightarrow Nat$
RecLevelCaller: $Nat \rightarrow Node$

This would require a change to all the currently defined transition rules. But as we always access only the current recursion level, we can solve this problem by defining *CRL* to be the default first argument of these four functions. Formally this is done by redefining the four macros *VALUE*, *ASSOC*, *LENGTH* and *CALLER* as follows:

$$\begin{array}{rcl} VALUE(n) & \triangleq & RecLevelValue(CRL,\ n) \\ ASSOC(n) & \triangleq & RecLevelAssoc(CRL,\ n) \\ LENGTH(n) & \triangleq & RecLevelLength(CRL,\ n) \\ CALLER & \triangleq & RecLevelCaller(CRL) \end{array}$$

This default argument allows us to reuse all transformation rules without changes.

Procedure and function call

In section 4.5 we described how *GuardOrCall* nodes as procedure calls can be differentiated statically from *GuardOrCall* nodes as type guards. For procedure calls we set the *NT*, *ActualParameter* (abbreviated as *AP*) and *Procedure* function as shown in picture 14 for a procedure or function call P(E1, E2, ..., En). The initial node of a procedure call is the initial node of the procedure and the terminal node is the ")"-node. Note that P can be an arbitrary complex designator that evaluates to a procedure declaration.

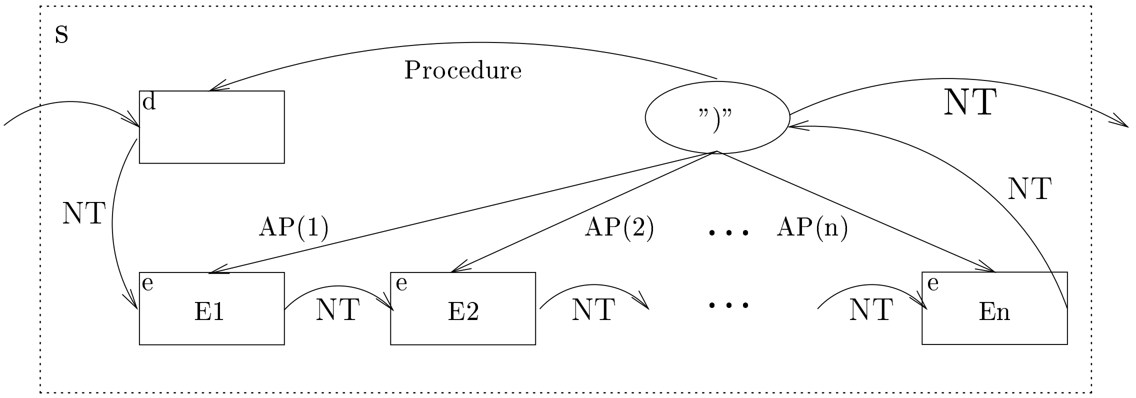

Figure 14: A procedure or function call

As the associated object is accessed by default on the current recursion level we lose contact with the variables outside a procedure if we increment the current recursion level by one. As only one procedure can be active on a certain recursion level, we can solve this problem by copying all associated objects, values and array lengths of the current recursion level to the next recursion level when we call a procedure.

if ")"(CT) **then**
    RecLevelValue(CRL + 1) := RecLevelValue(CRL)
    RecLevelAssoc(CRL + 1) := RecLevelAssoc(CRL)
    RecLevelLength(CRL + 1) := RecLevelLength(CRL)
    CRL := CRL + 1
    RecLevelCaller(CRL + 1) := CT
    CT:= CT.Procedure.VALUE
endif

The information is lifted to the new recursion level [10], as discussed before, the recursion level is incremented, the call stores itself as caller of the new recursion level, and control is passed to the procedure which is the value of the designator argument. The parameter and variables of the procedure overwrite the associated objects of the new recursion level, and the statements of the procedure can access all information on the new recursion level.

Return statement Procedures may return a value. These procedures are called functions and are characterized by the *ResulTypeId* in their declaration. A function call is an expression, and the body of the function has to contain a result statement.

ReturnStatement ::= RETURN[Expression]

If the optional expression is omitted, the return statement is equivalent to a procedure end task, and we include it in the universe *ProcedureEnd* (see below). The RETURN-node and the expression subtree are linked like a unary expression(e.g. the field Argument of the return node points to the terminal leaf of the expression).

if RETURN(CT) **and not** ProcedureEnd(CT) **then**
    RecLevelValue(CRL - 1, CALLER):= CT.Argument.VALUE
    CRL := CRL -1
    CT := CALLER.NT
endif

The return node assigns to the value of the caller the value of its argument, changes to the next lower recursion level and exits the function call. In order to make the result readable on the recursion level after the call, the result is assigned to the value field on that recursion level.

Procedure end In the last paragraph we added return statements without arguments to the universe *ProcedureEnd*. The transition rule for ProcedureEnd is last piece in the dynamic semantics of Oberon:

if ProcedureEnd(CT) **then**
    CRL := CRL -1
    CT := CALLER.NT
endif

[10] Note that we again use higher-order updates which have to be translated to ordinary updates.

5 Conclusions and Further Work

In this paper we present the dynamic semantics of Oberon as a mathematical model. This model can be used as the base for different proof calculi in order to verify Oberon programs or to embed the constructs of Oberon in an algebraic frame work, such that Oberon can be used as a back end for algebraic specification tools [29] [19].

On the other hand the model is an unambiguous description of Oberon. Due to its simplicity it can be used for standardization and education purposes. The mathematical training of an average undergraduate engineering student is sufficient to understand and apply the method.

An official Oberon standard based on this model would make portable software possible. It would be [15] "a comprehensive reference manual acting as an ultimate arbiter among possible interpretations of certain language features" both for programmers and compiler implementors.

An additional specification of the Oberon operating system [38], could be the base for correct and efficient low level software for safety critical systems. The flexibility of Evolving Algebras easily allows for an extension of the specification to Oberon dialects dealing with type bound methods [25], distributed computing [22]. ,and object oriented databases [32].

A Motivation of the more complicated record model

It is allowed to overwrite private fields of imported record types while extending then. This is unavoidable, because the implementor of the imported module has to be free in choosing the names of private record fields after one agrees on the "interface" defining only the exported record fields.

Suppose an Oberon module M1 contains an exported record type R1 with non exported field a of type A1. Further one imports M1 in a module M2 with a record type R2. One field of R2 has the name a and is of type A2 (figure 15). A record r2 of type R2 has now the following behavior: Two different fields of r2 have the name a, one with type A1 and the other with type A2. How can they be accessed? Suppose we have two pointers p1 and p2, the first of type POINTER TO R1 and the second of type POINTER TO R2. If we assign r2 to p2 the field with type A2 is accessible by designator p2.a. On the other hand if we assign r2 to p1 then we can access the field with type A1 by designator p1.f.

We model this situation in the store as follows. There are two Records o_1 and o_2 for r2 in the store. The first has type R1 and the function *Field(o1, a)* relates it with an object of type A1. The second has type R2 and *Field(o2,a)* relates it with an other object of type A2. An exported field of R, for instance b, references to an object which can be accessed by both *Field(o1,b)* and *Field(o2,b)*.

A type hierarchy gives now rise to an instance hierarchy of records. To keep the interconnection of this records we use a function

$$\_As\_ : Record \times RecordType \rightarrow Record$$

In figure 15 we illustrate function As with arrows, having the first argument as source and the second argument in the label as argument of As. The target of the arrow is as usual the value of the function As at this point.

- Oberon program:

```
MODULE  M1;
TYPE    P1* = POINTER TO R1;
        R1* = RECORD        a: A1;
                            b*: B
              END;
END M1.
```

```
MODULE  M2;
IMPORT  M1;
TYPE    P2 = POINTER TO R2;
        R2 = RECORD        a: A2;
                           c: C
             END;
END M2.
```

- Record r2: R2 and pointers p1: M1.P1, p2: P2 in the abstract memory:

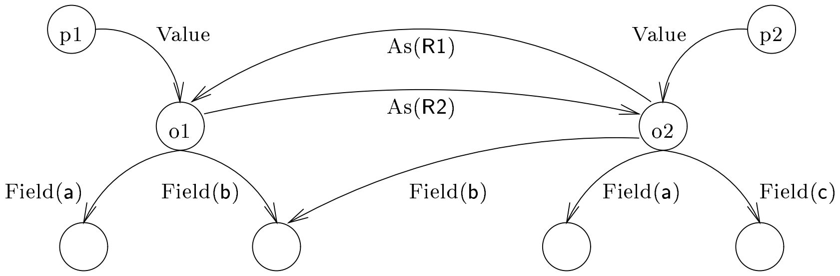

Figure 15:

B Syntax of Oberon

Program	::=	{Module}
Module	::=	MODULE ModObject “;” [ImportList]
		DclSequence
		[BEGIN StatementSequence]
		END ident “.”
ModObject	=	CodeObject
ImportList	::=	IMPORT Import {“,” Import } “;”
Import	::=	ImportObject [“:” ExternalName]
ImportObject	=	CodeObject
ExternalName	=	ident
DclSequence	::=	[CONST {ConstDeclaration “;”}]
		[TYPE {TypeDeclaration “;”}]
		[VAR {VarDeclaration “;”}]
		{ProcDeclaration “;” | ForwardDeclaration “;”}
ConstDeclaration	::=	ConstObject “=” ConstExpression
ConstObject	=	CodeObject
ConstExpression	=	Expression
TypeDeclaration	::=	TypeObject “=” Type
TypeObject	=	CodeObject
VarDeclaration	::=	VarObject { “,” VarObject } “:” Type
VarObject	=	CodeObject
ProcDeclaration	::=	PROCEDURE ProcObject FormalParameters DclSequence
		[BEGIN StatementSequence] ProcedureEnd ident
ProcedureEnd	=	END
ForwardDeclaration	::=	PROCEDURE “↑” ProcObject FormalParameters
ProcObject	=	CodeObject
FormalParameters	::=	[“(” [FPSection {“;” FPSection }] “)” [“:” ResultTypeId]] “;”
ResultTypeId	=	TypeIdent
FPSection	::=	VAR VarParamObject { “,” VarParamObject } “:” FormalType
		| ValParamObject { “,” ValParamObject } “:” FormalType
VarParamObject	=	CodeObject
ValParamObject	=	CodeObject
CodeObject	=	ident
Type	=	TypeIdent | GroundType | PointerType | ProcedureType |
		RecordType | ArrayType
TypeIdent	=	Designator

<table>
<tr><td>GroundType</td><td>=</td><td>BOOL | SHORTINT | INTEGER | LONGINT |
REAL | LONGREAL | SET | CHAR</td></tr>
<tr><td>PointerType</td><td>::=</td><td>POINTER TO ReferencedType</td></tr>
<tr><td>ReferencedType</td><td>=</td><td>Type</td></tr>
<tr><td>ProcedureType</td><td>::=</td><td>PROCEDURE
[“(”[ProcTypeParam {“,” ProcTypeParam}]“)”
[“:” ResultTypeId]]</td></tr>
<tr><td>ProcTypeParam</td><td>::=</td><td>[VAR] VirtualObject FormalType</td></tr>
<tr><td>VirtualObject</td><td>=</td><td></td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>RecordType</td><td>::=</td><td>RECORD [“(”BaseTypeId“)”]
[FieldDeclaration] {“;” [FieldDeclaration]} RecordEnd</td></tr>
<tr><td>BaseTypeId</td><td>=</td><td>TypeIdent</td></tr>
<tr><td>RecordEnd</td><td>=</td><td>END</td></tr>
<tr><td>FieldDeclaration</td><td>::=</td><td>FieldObject { “,” FieldObject } “:” Type</td></tr>
<tr><td>FieldObject</td><td>=</td><td>ident</td></tr>
<tr><td>ArrayType</td><td>::=</td><td>ARRAY Length OF ElementType</td></tr>
<tr><td>Length</td><td>=</td><td>ConstExpression</td></tr>
<tr><td>ElementType</td><td>=</td><td>Type</td></tr>
<tr><td>FormalType</td><td>=</td><td>OpenArray | TypeIdent</td></tr>
<tr><td>OpenArray</td><td>::=</td><td>ARRAY OF FormalType</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Designator</td><td>=</td><td>ident | Dereferenced | Qualified | Indexed | GuardOrCall</td></tr>
<tr><td>Dereferenced</td><td>::=</td><td>Designator“↑”</td></tr>
<tr><td>Qualified</td><td>::=</td><td>Designator “.” FieldSelector</td></tr>
<tr><td>FieldSelector</td><td>=</td><td>ident</td></tr>
<tr><td>Indexed</td><td>::=</td><td>Designator“[”Index“]”</td></tr>
<tr><td>Index</td><td>=</td><td>Expression</td></tr>
<tr><td>GuardOrCall</td><td>::=</td><td>Designator“(” [ExpList] “)”</td></tr>
<tr><td>ExpList</td><td>::=</td><td>Expression [“,” ExpList]</td></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Expression</td><td>=</td><td>SimpleExpression | Relation | TypeTest</td></tr>
<tr><td>Relation</td><td>::=</td><td>SimpleExpression RelOp SimpleExpression</td></tr>
<tr><td>RelOp</td><td>=</td><td>“=” | “#” | “<” | “<=” | “>” | “>=” | IN</td></tr>
<tr><td>TypeTest</td><td>::=</td><td>SimpleExpression IS TypeIdent</td></tr>
<tr><td>SimpleExpression</td><td>=</td><td>Term | SignedTerm | Sum</td></tr>
<tr><td>SignedTerm</td><td>::=</td><td>Sign Term</td></tr>
<tr><td>Sign</td><td>=</td><td>“+” | “-”</td></tr>
<tr><td>Sum</td><td>::=</td><td>SimpleExpression AddOp Term</td></tr>
<tr><td>AddOp</td><td>=</td><td>“+” | “-” | OR</td></tr>
<tr><td>Term</td><td>=</td><td>Factor | Product</td></tr>
<tr><td>Product</td><td>::=</td><td>Term MulOp Factor</td></tr>
<tr><td>MulOp</td><td>=</td><td>“*” | “/” | DIV | MOD | “&”</td></tr>
<tr><td>Factor</td><td>=</td><td>ExprInParentheses | NegatedFactor | Set | Designator |
intnumber | realnumber | character | string |
TRUE | FALSE | NIL</td></tr>
<tr><td>ExprInParentheses</td><td>::=</td><td>“(” Expression “)”</td></tr>
<tr><td>NegatedFactor</td><td>::=</td><td>“∼” Factor</td></tr>
</table>

StatementSequence	::=	Statement {“;”Statement}
Statement	::=	Assignment | NewStatement | ProcedureCall | IfStatement | CaseStatement | WhileStatement | RepeatStatement | LoopStatement | EXIT | WithStatement | ReturnStatement | NullStatement
NullStatement	=	
Assignment	::=	Designator “:=” Expression
NewStatement	::=	NEW“(”Designator“)”
ProcedureCall	=	Designator
IfStatement	::=	IF Condition THEN StatementSequence {ELSIF Condition THEN StatementSequence} [ELSE StatementSequence] EndIf
EndIf	=	END
WhileStatement	::=	WHILE Condition DO StatementSequence END
RepeatStatement	::=	REPEAT StatementSequence UNTIL Condition END
ReturnStatement	::=	RETURN[Expression]
CaseStatement	::=	CASE Selector OF Case {“|” Case} [ELSE StatementSequence] END
Selector	=	Expression
Case	::=	[CaseLabel{“,” CaseLabel}“:” StatementSequence]
CaseLabel	=	ConstExpression | LabelRange
LabelRange	::=	ConstExpression “..” ConstExpression
LoopStatement	::=	LOOP StatementSequence END
WithStatement	::=	WITH Designator“:” TypeIdent DO StatementSequence END

References

[1] I. Attali, D. Caromel, and M. Oudshoorn. A formal definition of the dynamic semantics of the eiffel language. *Australian Computer Science Communications*, 14-1:109 – 119, February 1993.

[2] Ch. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. An evolving algebra solution to the steam-boiler control specification problem. Technical report, Seminar on Methods for Specification and Semantics (Dagstuhl), June 1995.

[3] E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods*, pages 37 –51. Oxford University Press, 1995.

[4] E. Börger. Why use evolving algebras for hardware and software engineering. *LNCS*, 1012:236 – 271, 1995. SOFTSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics.

[5] E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-live processor out of basic components (the ape100 reverse engineering project). *Proc. First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145 – 148, 1996.

[6] E. Börger and I. Durdanović. Correctness of compiling occam to transputer code. *Computer Journal*, 1996. To appear.

[7] E. Börger, Y. Gurevich, and D. Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231 – 243. Oxford University Press, 1995.

[8] E. Börger and D. Rosenzweig. A mathematical definition of full prolog. *Science of Computer Programming*, 1994.

[9] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness. in: Logic Programming: Formal Methods and Practical Applications*, chapter 2, pages 20 – 90. series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[10] P. Borra, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. Technical Report 777, INRIA, Sophia Antipolis, 1987.

[11] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford University Press, 1995.

[12] Y. Gurevich and J.K. Huggins. The semantics of the c programming language. *LNCS*, 702:274 – 308, 1993. Selected papers from CSL'92.

[13] Y. Gurevich and J.K. Huggins. Errata to "the semantics of the c programming language". *LNCS*, 832:334 – 336, 1994.

[14] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 581, 1969.

[15] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2(1):335 – 355, 1973.

[16] J. Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247 – 293. Oxford University Press, 1995.

[17] G. Kahn. Natural semantics. In *Proceedings of the Symp. on Theoretical Aspects of Computer Science, Passau, Germany*, 1987.

[18] P.W. Kutter. Executable specification of oberon using natural semantics. Term Work, ETH Zürich, implementation on the Centaur System [10], 1996.

[19] P.W. Kutter. Integrating state charts in specware and aspects of correct oberon code generation. Master's thesis, Kestrel Institute, Palo Alto, September 1996.

[20] P.W. Kutter and A. Pierantonio. Montages: Towards the semantics of realistic programming languages. *JUCS*, 1997. submitted to the special issue of JUCS about Gurevich Abstract State Machines (evolving algebras).

[21] P.W. Kutter and A. Pierantonio. Montages: Unified static and dynamic semantics of programming languages. submitted to ICALP'97, 1997.

[22] S. Lalis and B. Sanders. Adding concurrency to the oberon system. *LNCS*, 782, 1994.

[23] C. Luginbühl. Denotationelle semantik von oberon. Master's thesis, ETH Zürich, 1995.

[24] R. Milner, Tofte M., and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[25] H. Moessenboeck. *Object-Oriented Programming in Oberon-2.* Springer Verla, second edition, 1995.

[26] D.A. Naumann. On the essence of oberon. *LNCS*, 782:313 – 327, 1994.

[27] M. Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, ETH Zürich, 1989.

[28] M. Reiser and N. Wirth. *Programming in Oberon.* Addison-Wesley, 1992.

[29] Y.V. Srinivas and R. Jüllig. Specware(tm): Formal support for composing software. Technical Report KES.U.94.5, Kestrel Institute, December 1994.

[30] Y.V. Srinivas and Juellig R. Augmenting algebraic specifications with structural subsorting. Technical report, Kestrel Institute, 1994.

[31] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* The MIT Press, 1977.

[32] J. Supcik and M.C. Norrie. An object-oriented database programming environment for oberon. In *Proc. Joint Modular Languages Conference (JMLC'9s), Linz, Austria*, 1997.

[33] USA University of Michigan at Ann Arbor. Evolving algebras web page. http://www.eecs.umich.edu/ealgebras/.

[34] C. Wallace. The semantics of the c++ programming language. In E. Bö"rger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.

[35] J. Welsh, W.J. Sneering, and C.A.R. Hoare. Ambiguities and insecurities in pascal. *Software Practice and Experience*, pages 685 – 696, 1977.

[36] N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35 – 63, 1971.

[37] N. Wirth. The programming language oberon. *Software - Practice and Experience*, 18(7):671 – 690, 1988.

[38] N. Wirth and J. Gutknecht. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[39] A.V. Zamulin. Algebraic specification of the programming language oberon. Technical Report 589.3945009.00005-01, Siberian Division of the Russian Academy of Sciences, 1995.