Montages: Towards the Semantics of Realistic Programming Languages

Philipp W. Kutter
Eidgenössische Technische Hochschule
CH–8092 Zürich (Switzerland)
kutter@tik.ee.ethz.ch

Alfonso Pierantonio
Università di L'Aquila
I–67100 L'Aquila (Italy)
alfonso@univaq.it

December 20, 1996

Abstract

We present a mathematical instrument called *Montages* for the formal semantic description of realistic, full–scale programming languages. Montages specifications are modular and their structure guides the design process. Syntax, static and dynamic semantics are described in a unified setting by means of visual and textual specification elements. The formalism is based on Gurevich's Abstract State Machines and can be equally well understood by language designers, compiler constructors, and programmers. Ease of specification, abstractness, compactness, and intuitivity make Montages suitable for the entire language live cycle, including documentation by language manuals. Compared with traditional manuals, Montages are similar in structure, length, and complexity, but with a formal semantics.

1 Introduction

Last two decades witnessed a large growth of language technology. New programming languages have been designed following the trend towards higher levels of abstraction, combining in most cases modularity and object orientedness with simplicity and efficiency. In fact, nowadays it is hard to envisage a programming language that does not support encapsulation in some form.

These languages normally required the efforts of large groups of persons and experience suggests that such languages not always are as compact and coherent as those realized by a single person, as for instance Pascal. Thus, new languages are defined passing through a number of stages from initial design to routine–use by programmers. These stages forming the programming language life cycle ([45]) are characterized as: requirement, design, specification, prototyping, implementation, testing, documentation, and maintenance. Designers need to keep track of the decisions which have been taken concerning language constructs, especially when the design is a team work. Exactly as for common software artifacts, precise, consistent and comprehensive descriptions are needed to communicate the intentions of the designer to the implementors, unless the designer and the implementor are the same person. At the moment, it is not unfair to state that the only comprehensive description of a programming language is usually its reference manual, which is mainly informal and open to misinterpretation. Formal approaches are therefore sought. Nevertheless, they caused a certain dissatisfaction concerning the pragmatic side of language specification since they were not always thought to support the language life cycle and tend to be very long and overly complicate.

In this paper we present a semi visual mathematical formalism, called *Montages*. Among the major features of Montages, we mention ease of specification, modularity, abstractness, compactness, and intuitivity which make it equally well understood by language designers, compiler constructors, and programmers. The syntax, static analysis and semantics, and dynamic semantics can be specified in a convenient way and the design is driven by the specification structure. The formalism is an extension of Gurevich's Abstract State Machines (GMs) [19, 5], which have been already successfully used for giving the dynamic semantics of languages like C [20] and many others (see [44, 7, 10, 8, 25], just to mention a few). These approaches while defining the dynamic semantics make the assumption that the control– and data– flow are already defined (an exception is the work in [7], as we mention later). This is a heavy limitation to the completeness of the given specifications and actually it is the main criticism against GMs, since it has been misinterpreted as a inherent problem of GMs. Control– and data–flow are crucial for the definition of the dynamic semantics of a language. Their definition provides the designer with enough insight to cope with later decisions concerning the language. Thus, starting already from the data– and control–flow means skipping all the stage of the static analysis and the checks of the context–sensitive constrains.

The contribution of Montages is twofold. On one hand, it suggests how to enhance the classical GM approach to the specification of programming languages with the formalization of the static analysis, i.e. how to define the data– and control–flow, and the static semantics. Section 2 shows how this task is crucial for the definition of the dynamic semantics. On the other hand, Montages provides a framework for the programming language life cycle for the type of engineering process being sought. The various steps of the engineering process serve as delineations of the language development activities. We sketch how Montages fit in the following stages

requirement, design, specification Once the requirements are established, the designer breaks the overall complexity into several chunks of behavior, each of them corresponding to a Montage. Such a "canonical" hierarchical decomposition is driven by the context–free grammar, which provides a useful mechanism for behavior localization and makes the design intellectually manageable. The description of the static analysis of a given construct provides useful insight about its dynamic semantics.

prototyping GM formalisms can be executed and there are already implemented interpreters [36, 15, 35]. At the moment, we are designing a tool, called *Darkroom*, which allows one by giving the Montages specification to obtain automatically a parser, a static semantics checker and an interpreter for that language.

documentation Language documentation – both internal and external – can be directly represented by the specification itself since it is very similar in structure, length, and complexity to language reference manuals.

maintenance Maintenance starts after the documentation and testing is done. It may result from late detection of flaws in all levels of the life–cycle: forgotten or additional points in the requirement, bad design decisions, specification flaws, misinterpreted prototypes, buggy implementation, not enough testing, ambiguous documentation, or errors made during earlier maintenance. Maintenance as the correction of these errors introduces not new tasks, but depends on the flexibility and reusability of the results in all previous stages. Montages provides such flexibility in different measures. In particular, location of behavior is obtained in a canonical way, addition of new constructs or modification of old ones is obtained defining new Montages or performing limited modification in very neat drawn boundaries of behavior. Interdependencies between different Montages (especially needed for the static semantics) are defined formally by means of inter–level arrows.

In the paper, we try to provide evidence of what we claim by means of a relevant case study: the Montages specification of a modern language like *Oberon* [46, 39]. We approached the language in three subsequent steps to stress certain qualities of the method. First, how the features of Oberon can be specified independently and how the outcome of this stage is good for documentation. Second, how a complete sub-language is used for early prototypes which allow to understand small, manageable sub-sets of large languages. Finally, the completion of the language with complex features shows how smoothly new features can be integrated in existing specification parts, which enhances flexibility of the method.

The paper is organized as follows. Section 2 introduces Montages through a simple yet complete expression language. Here besides preliminary notations the basic machinery is illustrated in a step by step manner. Section 3.3 presents a complex Montages specification of a non–toy language, i.e. Oberon. Finally in section 4 some conclusions are drawn.

2 Montages

In this section we present *Montages* [29], a method for defining programming language semantics using Gurevich Machines[1]. GMs have been proposed by Y.Gurevich as a dynamic generalization of multi–sorted algebras, intended to provide a more versatile notion of Turing machine, "able to simulate arbitrary algorithms in a direct and essentially coding-free way" [19]. In short, GMs are a state–based formalism in which a state is updated in discrete time steps. Unlike most state based systems, the state is given by an algebra, i.e. a collection of universes and functions. The state transitions are given by so called *transition rules* that transform state (algebras) in such a way that universes are extended with new elements and functions are modified pointwise.

GMs have already been successfully used to model the dynamic semantics of programming languages such as C [20], C++ [44], Oberon [25], Occam [7], Prolog [10], and VHDL [8]. These works are based on control and data–flow, but most of them do not formalize the generation of the flow information. This has been the major criticisms against these works, and some people started to misinterpret this situation as a general flaw of GMs. One exception to this practice is the work on Occam [7, 6]. The solution chosen there is unfortunately not directly exploitable to give static semantics.

Montages suggests how static analysis and semantics of programming languages can be given together with their dynamic semantics in the same setting. The way control and data flow information is collected allows for a specification of statics semantics as surprisingly simple predicates.

The formalism makes use of a (semi) visual language that allows unified and coherent specification of syntax, static semantics and analysis, and dynamic semantics. The static aspects of Montage specifications resemble data and control flow graphs, and the overall specifications are similar in structure, length, and complexity to those found in common language manuals. Montages is a formal instrument which can be equally well understood by language designers, compiler constructors, and programmers.

[1] Gurevich Machines were formerly called Evolving Algebras.

This section is organized as follows. In section 2.1 we present the formal semantics of Montages along a simple example. The complete specification of syntax, static analysis and semantics, and dynamic semantics of a small expression language, and the mathematical model corresponding to such a specification are given. Important features of Montages, that are not used in this example are presented in section 2.2.

2.1 Basic Montages

A Montages specification of a language $\mathcal{L}$ defines a Gurevich Machine $\mathrm{M}_{\mathcal{L}}$ that for a given program $\mathcal{P}$ of $\mathcal{L}$ analyses and defines the control– and data–flow, checks the static semantics, and in turn executes the dynamic semantics. The transition rule of $\mathrm{M}_{\mathcal{L}}$ is applied to an initial state $\mathrm{I}_{\mathcal{P}}$, which depends on the program $\mathcal{P}$: for each different program, there is a different initial state.

In the sequel, we first illustrate how an initial state is defined based on the syntax of a program, and then how the machine $\mathrm{M}_{\mathcal{L}}$ models the statics and the dynamics of the language $\mathcal{L}$.

2.1.1 Initial State $\mathrm{I}_{\mathcal{P}}$ — Syntax

The initial state $\mathrm{I}_{\mathcal{P}}$ encodes the syntax of a program $\mathcal{P}$. $\mathcal{L}$-programs are terms generated by a context free grammar $\mathrm{G}_{\mathcal{L}}$. Each program generated by $\mathrm{G}_{\mathcal{L}}$ can be represented as a *compact derivation tree.* The initial state $\mathrm{I}_{\mathcal{P}}$ associated with a program $\mathcal{P}$ is given by universes and functions representing the nodes and branches of the compact derivation tree. The functions are called *selector functions* since they allow to select the descendants of a node. A compact derivation tree is a parse tree in which a node n together with its single descendant d is collapsed to one node, having both the labels of n and d and having only the descendants of d.

Figure 2 contains a context–free grammar of an example expression language $\mathcal{L}$, with non–terminals for *Expression*, *Sum*, *Term*, *Product*, *Factor*, and *ExprInParenthesis*, a term $\mathcal{P}$ of that grammar, the normal derivation tree of $\mathcal{P}$, and its compact version. The sign "=" is used in the grammar instead of "::=" whenever a rule generates only a unique descendant. As pointed out above, in the compact derivation tree this descendants is collapsed with its parent. These rules are called *synonym productions.* All the following is based on compact derivation trees.

As we said the initial state $\mathrm{I}_{\mathcal{P}}$ contains the compact derivation tree of the program $\mathcal{P}$. The state $\mathrm{I}_{\mathcal{P}}$ has an universe, called *Node*, which contains both the non–terminal and terminal nodes of the tree. Moreover, we have a sub-universe of *Node* for each non–terminal and terminal symbol, in which are contained the nodes belonging to that category. The universes of the example in figure are the following, where the naturals identifies the nodes of the compact derivation tree:

$$\begin{aligned}
Expression &= \{\mathit{1}, \mathit{2}, \mathit{5}, \mathit{12}, \mathit{14}\} & Sum &= \{\mathit{1}, \mathit{2}, \mathit{12}\} && (1)\\
Term &= \{\mathit{4}, \mathit{5}, \mathit{7}, \mathit{8}, \mathit{14}, \mathit{16}\} & Factor &= \{\mathit{4}, \mathit{5}, \mathit{8}, \mathit{10}, \mathit{14}, \mathit{16}\} && (2)\\
ExprInParenthesis &= \{\mathit{10}\} & Product &= \{\mathit{7}\} && (3)
\end{aligned}$$

$$Ident = \{\mathit{4}, \mathit{5}, \mathit{8}, \mathit{14}, \mathit{16}\} \qquad (4)$$

$$\begin{aligned}
\text{“*”} &= \{\mathit{9}\} & \text{“+”} &= \{\mathit{6}, \mathit{3}, \mathit{15}\} && (5)\\
\text{“(”} &= \{\mathit{11}\} & \text{“)”} &= \{\mathit{13}\} && (6)
\end{aligned}$$

S-Expression : *Sum* $\longrightarrow$ *Expression*	$1 \longmapsto 2$ $2 \longmapsto 5$ $12 \longmapsto 14$
S-Expression : *ExprInParenthesis* $\longrightarrow$ *Expression*	$10 \longmapsto 12$
S-Term : *Sum* $\longrightarrow$ *Term*	$1 \longmapsto 4$ $2 \longmapsto 7$ $12 \longmapsto 16$
S-Term : *Product* $\longrightarrow$ *Term*	$7 \longmapsto 8$
S-Factor : *Product* $\longrightarrow$ *Factor*	$7 \longmapsto 10$
S-"+" : *Sum* $\longrightarrow$ "+"	$1 \longmapsto 3$ $2 \longmapsto 6$ $12 \longmapsto 15$
S-"*" : *Product* $\longrightarrow$ "*"	$7 \longmapsto 9$

Figure 1: Selector functions

Please note how the synonym productions cause some nodes to belong to more than one universe. Together with the universes, in the initial state $\mathrm{I}_{\mathcal{P}}$ there are also the selector functions which link the nodes downwards according to the tree as one would expect. Since universes are not disjunct these functions are overloaded, i.e. the same selector function is used to select descendants from more than one category of nodes. The definition of the selector functions are given in figure 1. In this figure we list only those functions which are used later. The notation S- stays for "selector" and it is used for distinguishing the function names from the universe ones.

The definition of $\mathrm{I}_{\mathcal{P}}$ required only the grammar part $\mathrm{G}_{\mathcal{L}}$ of the specification of the language $\mathcal{L}$. Now we illustrate how the static analysis of $\mathcal{L}$ is specified using Montages and how the GM semantics of this specifications generates the control– and data–flow graph of $\mathcal{P}$.

2.1.2 Transition rules — Semantics

The techniques used for modeling the static analysis and the dynamic semantics is inspired by the classical use of GMs for language specifications [20, 44, 25]. These approaches provided the specification method with only dynamic semantics. The static analysis information (control and data flow) are supposed already present in the initial state, and the program is supposed to adhere to the static semantics constraints.

In order to formalize the static analysis and semantics we extended the method in a uniform way, retaining the advantages of the previous works. These approaches operate on the leaves of the parse tree, which correspond to the tokens of the program text. Each token is interpreted as abstract instruction. We therefore refine the transition rule of $\mathrm{M}_{\mathcal{L}}$ in such a way that:

- it first traverses the tree, characterizing the static semantics, stores the results of the control– and data–flow analysis as attributes (unary functions) of the leaves, and checks the static semantics;

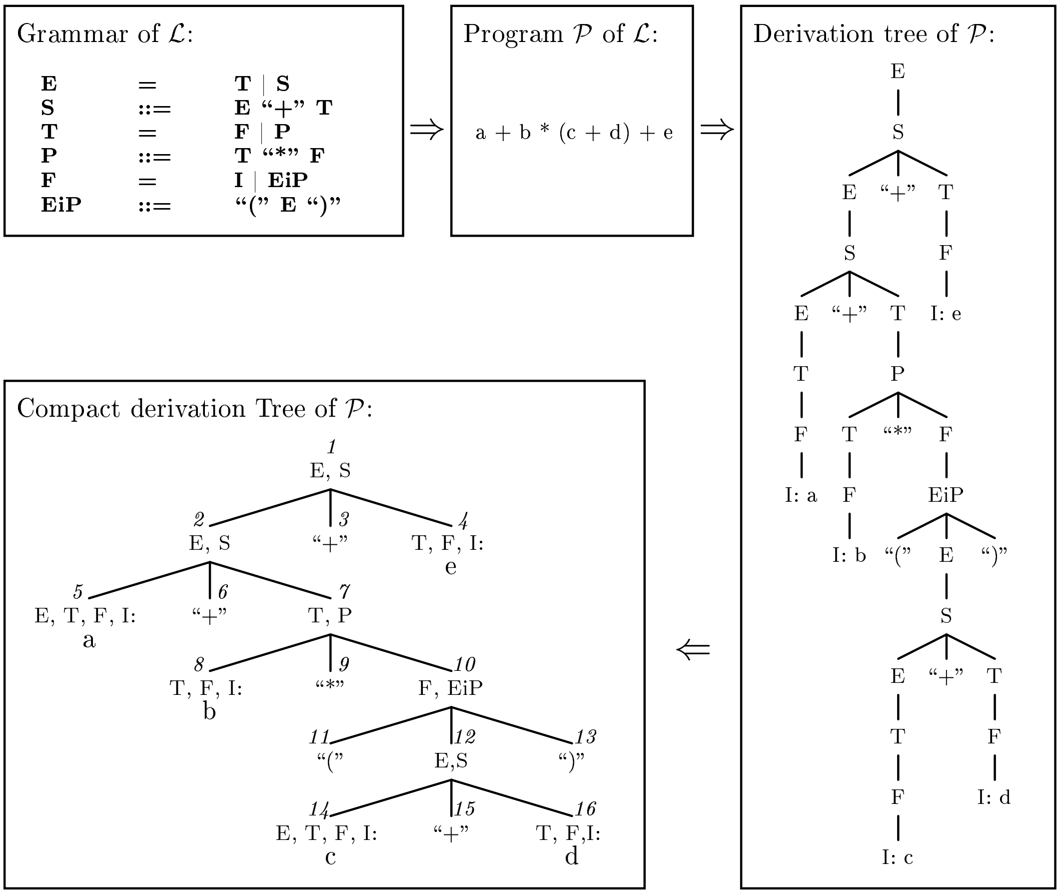


Figure 2: Example for a grammar, a program, and its derivation trees

- then, using these attributes, is able to operate on the leaves of the parse tree, characterizing the dynamic semantics.

We see now in the following how static analysis and dynamic semantics are formalized in Montages.

Static Analysis and Semantics The fact that the dynamic semantics operates only on the leaves of the tree, which correspond to the tokens of the program text, is a very important aspect of the overall methodology. In fact, the control– and data–flow information is provided as attributes of the tokens. The attributes can be illustrated as labeled arrows between the tokens. As convention, we use dotted arrows to denote control–flow and solid arrows to denote data–flow[2]. An addition token "+", for instance, has typically two data–flow attributes *Left* and *Right* pointing to the tokens representing its left and right arguments. This is illustrated in the control and data flow of $\mathcal{P}$ given in figure 3, which contains the control– and data–flow of the program $a+b*(c+d)+e$. Please note as well the NT–labeled control–flow arrows, which define the next task attribute in the control–flow. As in traditional control– and data–flow graphs, the flow graph links directly tokens of the program text, and does not use internal nodes of the derivation tree.

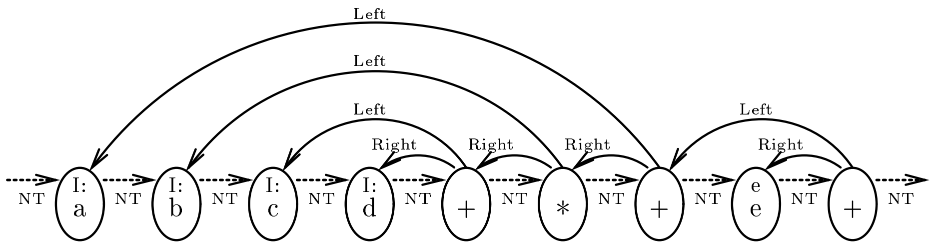


Figure 3: Control and data flow of $\mathcal{P}$

The control– and data–flow graphs are defined by performing the static analysis on the tree. The difficulty is to specify this static analysis. The result of the analysis must be a data– and control–flow structure between the leaves of the derivation tree as shown.

Montages provides a method for specifying static analysis, i.e. for defining control– and data–flow starting from the syntax of the language. The parse tree is made of recurrent patterns, since it is generated by a grammar. Each production rule

$$n ::= E$$

has different occurrences in the tree, in particular we can imagine the non–terminal n matching subtrees whose descendants are structured according to the right–hand side E of the production. The root of such a subtree is a node corresponding to the left–hand side n. Non–terminal (terminal) symbols in E representing this view the direct descendants of that root. Linking together this symbols results in flow arrows between internal nodes of the derivation tree. In order to get normal flow graphs we have to move this arrows to the leaf level. We found a heuristic, which we will describe later, that does this task in a satisfactory way. Thus we can define control– and data–flow graphs by specifying how the symbols in the grammar production – both terminals and non–terminals – should be linked together.

[2]The reader familiar with data–flow techniques will remark in the following that our data–flow arrows go in the opposite direction of the arrows in common data–flow graphs. If one uses our approach to generate data and control flow graphs, the data arrows of the presented examples must be inverted.

An example are the Montages of $\mathcal{L}$ in figure 4. There we see how the already mentioned arrows *Left*, *Right*, and NT are defined between symbols of the right–hand side of the corresponding (written just above the respective graph) production rule.

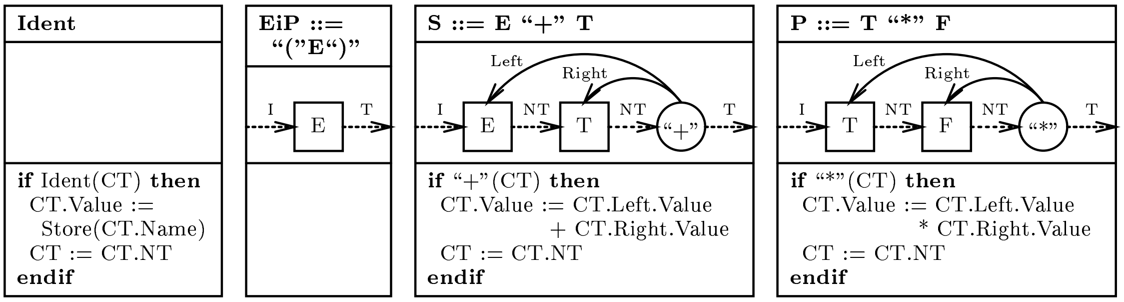


Figure 4: Montages of $\mathcal{L}$

A Montages specification consists of a set of modules associated with the production rules. A module, called Montages, contains four parts: the first is the production rule the Montage speaks about; the second contains the graph defining control– and data–flow as discussed. The third, marked with keyword **condition** contains the static semantics condition, using the flow information. And the last box contains the dynamic semantics definitions for the tokens introduced by the production rule. Please note that the Montages of $\mathcal{L}$ in figure 4 do not contain the third part.

We did use the convention that a inner node represents one of its leaf–descendants, but we have not answered the question which of them! As a first step, we define two special leaves per node, that can be used as its placeholder. These leaves are called the *initial* and the *terminal leaf* of a node n, since conceptually they are the first respectively the last leaves in the control–flow of the leaves of n.

The initial and terminal leaves are defined recursively for all nodes of the derivation trees. The specification of that definition is given graphically: for each production rule, the static analysis graph contains a dotted arrow from the box border to a symbol, called the *I–symbol* of the production rule, and a dotted arrow from a symbol, called the *T–symbol* of the production rule, to the border of the box. By giving I– and T–symbols for each rule, we define the functions $Initial : Node \rightarrow Leaf$ and $Terminal : Node \rightarrow Leaf$ as follows. We use the notation l.Terminal instead of Terminal(l).

- For each leaf l:

$$l.\text{Terminal} = l \qquad \text{and} \qquad l.\text{Initial} = l$$

- For each inner node o labeled with non–terminal n, provided that the production rule

$$n ::= E$$

generated a direct descendent i of o for the I–symbol of the rule, and a direct descendent t of o for the T–symbol of the rule:

$$o.\text{Initial} = i.\text{Initial} \qquad \text{and} \qquad o.\text{Terminal} = t.\text{Terminal}$$

If we apply this definition to the compact derivation tree in our example, we get the situation in figure 5. Please note that in this picture dotted and solid arrows have nothing to do

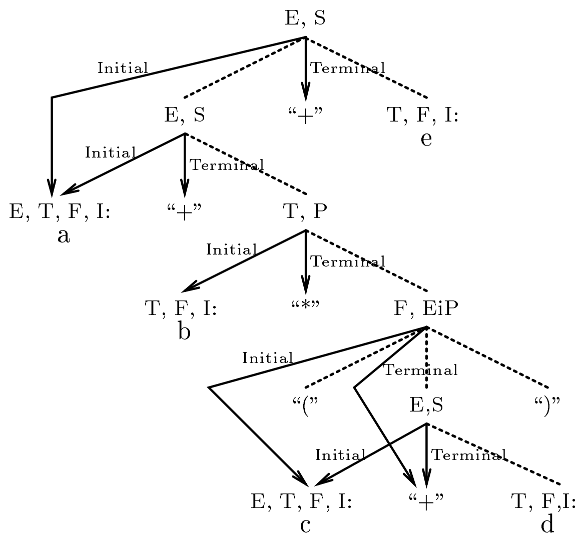


Figure 5: Definition of Initial and Terminal in $\mathcal{P}$

with control and data flow. The dotted arrows are the tree structure, and the solid arrows define initial and terminal nodes.

Based on the definition of initial and terminal leaf, we can now define which leaves are linked by control and data arrows as follows:

- A *dotted control–flow arrow* links the terminal leaf of its source with the initial leaf of its target. This choice is based on the heuristic that control flow typically is some kind of sequential control–flow and therefore an arrow from inner node a to inner node b means that after processing the last (terminal) task of a, one has to process the first (initial) task of b.

- A *solid data–flow arrow* links the terminal leaf of its source with the terminal leaf of its target. The rationale is the following: assume, for instance, an inner node representing an expression; the value of the expression can be evaluated, only if all its sub-terms have been evaluated, i.e. they had control. Thus if the control is given to the terminal leaf of the expression, the result is available, and can be stored as field of that leaf. This and other typical situations show, that the terminal leaf is well suited as access point for data-flow.

Using the definitions, we can understand how the four Montages in figure 4 define the control– and data–flow graph of figure 3. The start point is the compact derivation tree of $\mathcal{P}$ with the definitions of Initial and Terminal, as given in figure 5. Then for each inner node n in the tree do the following:

- Find the Montage having one of the labels as left–hand side of its production rule. It is guaranteed that there is only one of such a Montage.

- Draw the arrows specified in the graph of that Montage between the descendants of n.

- Change each of these arrows to an arrow between leaves as follows:
  - a control arrow from node a to a node b is changed to an arrow from the terminal leaf of a to the initial leaf of b;

- a data arrow from node a to a node b is changed to an arrow from the terminal leaf of a to the terminal leaf of b.

Some final remarks:

1. A symbol x in the production rule is represented in the graph by a box or circle labeled with x. Alternatively we use the label S-x, or if a symbol x appears more than once in the production rule, its occurrences in the graph are labeled with S1-x, S2-x, etc. Formally x is the name of a universe containing all nodes labeled with x and S-x (respectively S1-x, S2-x, ...) is a function that links the parent node with its descendant (respectively descendants) generated by the symbol x in the right–hand side of the corresponding production rule, as we already mentioned

2. The control–flow part of the flow graph consists of the leaves and solid arrows, and the data–flow part of the flow graph consists of the nodes and dotted arrows. Traditionally these parts are drawn as two separate graphs.

3. The formal definitions given in [29] define an operational graph traversal that implements the definition given here. One could think about a static definition of the up to now discussed static parts as well. The disadvantages are, that we need additional static formalisms, for instance Natural Semantics [23] for that. The formal definition as given needs only Gurevich machines, which makes it more accessible.

4. For simplicity we modeled a fixed sequential execution order of the arguments. Non-deterministic and even parallel order can as well be modeled with Montages [].

Up to now we explained the static aspects of Montages specifications, i.e. we described how to enrich the grammar for specifying the static analysis. Usually Montages have also static semantics constraints, which are first–order formulas. Such formulas can be evaluated after the data–flow has been performed. The Montages in figure 4 have no static semantics condition. Later in this section we will give additional information needed for the use of static semantics conditions and section 3.3 abounds of examples for such conditions.

Dynamic Semantics The fourth part of a Montage is the dynamic semantics part. As mentioned this part is given by means of GM transition rules.

A transition rule is a description of how the current state evolves starting from the initial one. The initial state for the dynamic semantics transition rules in Montages is the control– and data–flow graph. The transition rules are described by means of *mathematical microcode* based over the following set of constructs: *update*, *block* , *conditional*, *range* and *extend*. The update operator is a pointwise modification of a function. Its synopsis is

$$f(t_1, \ldots, f_n) := t_0$$

Such an expression first evaluates the terms $t_1, \ldots, t_n$ and t_0 over the current state to the elements $e_1, \ldots, e_n$ and e_0, and then modifies f on the point $(e_1, \ldots, e_n)$ to e_0. The update of f does not affect the definition of the function over the rest of the domain. A set of transition rules constitutes a block and the rules in the block are executed in parallel. A conditional rule is written as follows

if *cond* **then**
    rule
endif

where *cond* is a first order predicate, and *rule* a transition rule. This rule is triggered if the value of the guard expression *cond* is evaluated to *true*. A conditional rule may have also an *else* branch, in such a case the corresponding transition rule is triggered if the guard is not evaluated to *true*. The operators range and extend are explained as they are encountered in the paper.

Given an initial state, the different states of an GM are reached by iteratively triggering the transition rule until the GM reaches a state which cannot evolve anymore. The initial state of the dynamic semantics is the result of the static analysis. The transition rule characterizing the dynamic semantics is the block of all transition rules given in the lowest part of the Montages. In the following we describe the typical structure of such rules for sequential languages. Some non-sequential languages need a slightly different form of rules.

A sequential program typically runs by executing one *machine instruction* after the other. The current instruction is pointed by a "program counter". We already mentioned that the leaves are considered as the instructions in our model, therefore we call them tasks, if we speak about dynamic semantics. As "program counter" we use a dynamic nullary function *CurrentTask*, abbreviated with CT.

$$CT : Leaf$$

This function points always to the task which currently has the control. Typically in each step CT is set to the next task in the control flow, which can be retrieved as the NT attribute of the task. In each Montages of $\mathcal{L}$ we find in the dynamic semantics part the update

CT := CT.NT

which sets the current task to the next task. An other common property of the rules is, that they have the form

if "terminal"(CT) **then**
 ...
endif

The guard of such a rule is evaluated to true, if the current task is labeled with "terminal". Thus the different rules indicate what is executed if a leaf with the corresponding label has the control. For all types of tasks in $\mathcal{L}$ (e.g. Ident, "+", and "*" tasks) we see that the field *Value* of the current task is updated. If the current task is an Ident, then the value of the store for that variable is assigned to the *Value* field

CT.Value := Store(CT.Name)

In this simple example, the store is modeled as a function from variable names to integers. The micro syntax of the identifiers is stored in an attribute

$$Name : Ident \rightarrow String$$

If the current task is an addition (respectively a multiplication) operator, then the *Value* field of the current task is set to the sum (respectively product) of the *Value* field of the left task in the data–flow and the Value field of the right task in the data-flow.

We can now execute the example $\mathcal{P}$ in figure 3. First we need to define the values of the functions CT, and Store in the initial state. We set CT to the a–task and Store to {(a, 4), (b, 8), (c,3), (d, 7), (e,9)}. The initial state of the system consists of the definitions of the functions CT, Store and the control– and data–flow graph. The repeated execution of the transition rules will now result in the calculation of 4 + 8 *(3 + 7) + 9. In the first step, the *Value* field of the a–task is set to 4 and parallelly, the current task is set to the b–task. In the second, third, and fourth step, the *Value* field of the b–, c–, and d–tasks are set to 8, 3, and 7. After those steps, the current task is equal to the first "+"–task. According to the transition rule for "+"–tasks, the *Value* field of that task is set to the sum of the *Value* fields of the c– and d–tasks and the current task is set to the "*"–task. This task multiplies the *Value* fields of the first "+"–task and the b–task, and stores the result as its own *Value* field. This process continues, until the last "+"–task sets the current task to undefined, since the NT attribute of the last "+"-task is not defined. At this point the system terminates, no update can be triggered any more. If we made a correct protocol, the *Value* field of the terminal "+"–task holds the result of the calculation: 93

2.2 Additional Features

Montages are modular specifications in which the behavior is highly factorized and localized. In order to support as much as possible the language design process we introduced some additional features which cope with specific aspects of the specification stage. In particular, we present

- lists management, for a convenient and abstract treatment of lists of block in programming language, e.g. a sequence of type declarations;
- hierarchical montages, for specifying attributes which link different montages;
- extra transition rules, for describing *ad–hoc* situation for the static analysis.

With the last point we guarantee that we can use the full power of GMs, not only for dynamic but as well for static aspects. Like this we can describe ad–hoc any new situation, without introducing ad–hoc features in the formalism.

2.2.1 Lists

In many approaches a major part of a language specification is concerned with the processing of lists. Therefore we decided to include in Montages a simple, yet powerful list model together with graphical and textual specification elements that can be used to avoid all explicit list processing. If the right–hand side of a production rule contains a symbol in a { } part, a list of descendents is generated.

As access point for that list, there is one node generated, a so called *list node*, that provides access to all needed informations about the list. An attribute *ListLength* of the list node is set to the length of the generated list and a binary mix–fix function

$$\_[\_] : \mathit{ListNode} \times \mathit{Nat} \rightarrow \mathit{Node}$$

can be used to retrieve the elements of the list. The initial and terminal leaves of a list node are defined to be the initial leaf of the first element, respectively the terminal leaf of the last element in the list. If the list is empty, they point to the list node itself, which then serves as dummy element. The dynamic semantics of that dummy element corresponds to the skip command. For convenience, we assume that the pattern

s { "t" s },

where “t” is an arbitrary terminal, generates just one list node, as if the pattern would be {s}.

In static analysis graphs, a list node is represented by a box, which is marked in the right–top corner with the keyword LIST. A second box or circle within the LIST–box represents the single elements in the list. An arrow from a node within a LIST–box corresponds therefore to a family of arrows from all of the elements in that list. As example we take the upper Montage in figure 9 that is used later in the paper for variable declarations. The production rule

VarDeclaration ::= VarObject {“,” VarObject} “:” Type

generates two descendants, a list node for the list of VarObjects and a node labeled with Type. As defined before, the formalism recognizes the pattern

Var {“,” Var }

and generates only one list node for it.

A data–flow arrow *StaticType* must link each VarObject in the list with the terminal leaf of the Type. This is in Montage 9 graphically specified by drawing a data–flow arrow from the circle within the list box, which represents the single list elements, to the Type box.

Experience showed that often binary functions which link a leaf with all elements of a list are used. The second argument of such a function is the position in the list. An example where this is needed are the actual parameters of a procedure call. We allow to define this type of binary function graphically by means of a solid arrow going into a list box. As example we specify the control and data flow of a procedure call with parameters

ParamProcCall ::= Call “(” [Expression
{“,” Expression}] “)”

The the AP arrow in the Montage given in figure 6 defines a function ActualParameters

$$AP : Call \times Nat \to Task$$

which links the call task with its actual arguments.

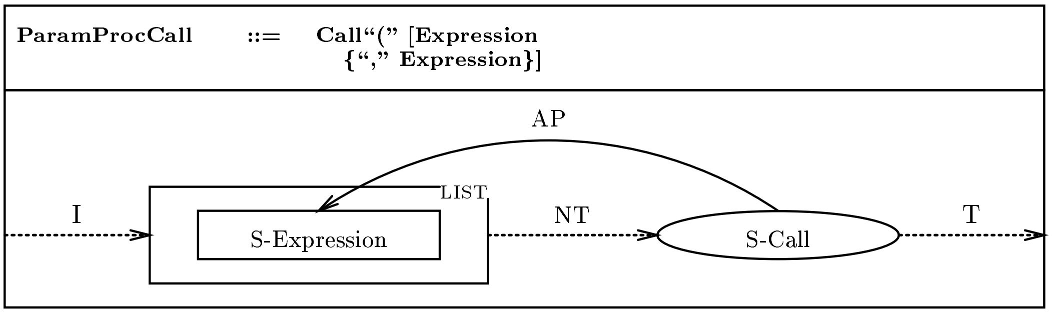


Figure 6: Example for an arrow into a list box

2.2.2 Inter-level arrows

Montages, as they have been presented, allow attributes to be defined only inside one Montage. Sometimes it can be convenient to have attributes ranging over two different Montages. Therefore we introduce inter-level arrows, that allow to define links over more than one level graphically. In order to draw inter-level arrows, we need to represent in the graph not only the direct descendants, but as well the lower level descendants.

In the Montages graph, the surrounding box represents the parent of the direct descendants represented by the boxes and circles in the graph. This image can be used to represent descendants on the next lower levels: they are drawn as small boxes or circles within the box representing their parent node. Like this we can represent descendants on arbitrary lower levels. As illustration we give an alternative expression language in Polish form, which results in a shorter grammar, not needing parentheses:

Expression	=	Ident | BinExpr
BinExpr	::=	Operator ExprPair
ExprPair	::=	Expression Expression
Operator	=	"+" | "*"

In the Montages we will again use short forms for the symbols, E for Expression, I for Ident, B for BinExpr, P for ExprPair, and O for Operator. In this language the example arithmetic expression $a + b * (c + d) + e$ is denoted by

$$+ + a * b + c\ d\ e$$

Please note that this syntax is perfectly parsable by a computer. We want to define the control–, data–flow, and dynamic semantics of this language as for $\mathcal{L}$, e.g. the generated control– and data–flow of $+ + a * b + c\ d\ e$ should look like in figure 3, and the the dynamic semantics rules should be reusable. The Montages specification of the new languages in figure 7 fulfills this requirements.

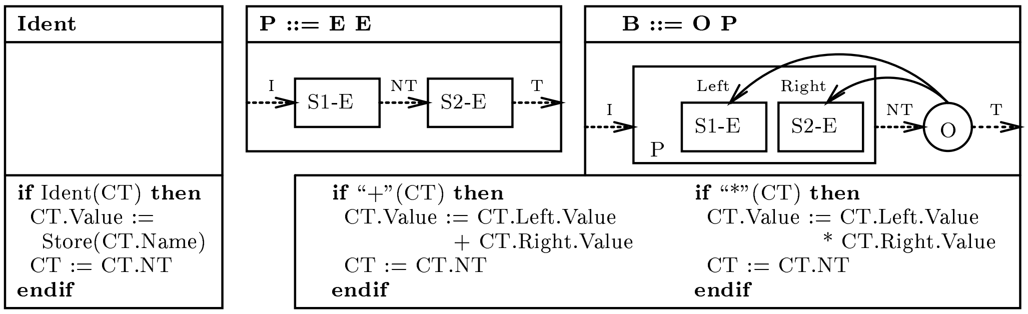


Figure 7: Montages specification with inter-level arrows

The Montage for BinExpr (B) contains two inter-level arrows labeled Left and Right, and uses the explicit selector function syntax for S1-E and S2-E. The inter-level arrows link the B node with the left and right expression, which are going to be generated by the P rule. In the P Montage, only the local arrows between the right-hand-side symbols of P are defined. The boxes S1-E and S2-E represent in the B Montage not direct descendants of a B node, but descendants of the B-descendant P. Inter–level arrows between boxes and circles representing such lower level nodes have the same semantics as normal arrows, they connect initial and terminal leaves according to the conventions. The Left arrow in Montage B, for instance, links the (terminal) of the O-node to the terminal of the corresponding expression, since it is a data–flow arrow.

An interesting situation is, if inter–level arrows point in sub–nodes of list nodes. This situation is for example given in the RecordType Montage in figure 18. There a control–flow arrow *Parent* has as source the sub-node S-FieldObject and as target the node S-RECORD. The source node S-FieldObject is inside a list box, which is a sub–box of the box S-FieldDeclaration which is inside another list box. According to the list box and sub box semantics, this means that this arrow stands for a family of arrows defined as follows. For each element d in the field declaration list, there is an arrow from each element in the list–descendant FieldObject list–descendant of d. The target for all of these arrows is the S-RECORD node.

2.2.3 Adding Transition Rules

Up to now, we could describe the semantics of Montages in a way that is completely independent from the order in which static analysis is executed. The only thing we said was, that for each inner node in the compact derivation tree, the static analysis must be performed. The definition was in this respect completely static, it can for instance be formalized as a set of axioms. Since we use GMs for the semantics of the dynamics, we prefer to use GMs as well for the static analysis. As mentioned, GMs are actually the only formalism needed for the formal semantics of Montages. In this section we start giving the GM-semantic of the static analysis, and show how we can add textual elements in the static analysis part of a Montage. At the end of the section we mention shortly how the GM-semantics of Montages traverses the tree, checks the static semantics, and performs the static analysis. More details on this process are found in [29].

In the GM semantics of Montages the performance of static analysis corresponds to the execution of a transition rule. An arrow in the static analysis graph corresponds to an update, and each arrow into a list box corresponds to range of updates. The block of these updates is the formal semantics of the static analysis. In the following we show how this block is defined, and how we can add arbitrary additional textual rules to the graphs.

The transition rules in the static analysis part must denote the descendants of a node textually instead of graphically. Lets assume that a Montage with production rule $n ::= E$, and a function

$$CN : Node$$

which points to an n node. The descendants of CN are represented graphically by the boxes and circles labeled with the symbols in the right-hand-side E. Textually, the descendants can be accessed by means of the selector-functions which we mentioned in section 2.1.1. Formally the selector-functions are defined as follows:

- If E is of the form $s_1 s_2 \ldots s_m$ we access the m descendants nodes as CN.S-s_1 ... CN.S-s_m. If the same symbol s occurs more than once in $s_1 s_2 \cdots s_m$, we enumerate the attributes from left to right: CN.S1-s denotes the first s-descendant, CN.S2-s denotes the second and so on.

- If E contains a symbols s in a { } part, then CN.S-s delivers an element of a universe *ListNode*. As noted in section 2.2.1 a binary mix–fix function

$$\_[\_] : ListNode \times Nat \rightarrow Node$$

maps this list node and a natural i to the s–node at position i in the list of s–descendants produced by the s in the { } brackets. A function

$$ListLength : ListNode \rightarrow Nat$$

yields the length of the list.

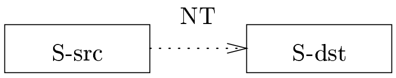
- For convenience, we assume that the pattern

$$s\{\text{“t”}\ s\}$$

where t is an arbitrary terminal, is treated as if the pattern would be $\{s\}$.

Of course the selector functions can as well be used to access textually descendants of descendants, as inter-level arrows do graphically. Using this notation, We can give the update defined by an arrow in the static analysis graph.

- Each dotted data–flow arrow as the following

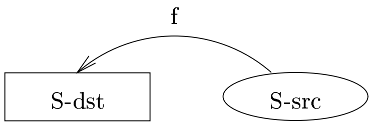


defines the update

CN.S-src.Terminal.NT := CN.S-dst.Initial

This corresponds to the earlier definition that control flow links the terminal leaf of the source with the initial leaf of the target.

- Each solid data–flow arrow as the following



defines the update

CN.S-src.Terminal.f := CN.S-dst.Terminal

This corresponds to the earlier definition that data flow links the terminal leaf of the source with the terminal leaf of the target.

Arrows out of list boxes correspond to one arrow for each element in the list. The transition rule corresponding to the StaticType arrow in Montage 9, which has been explained in section 2.2.1, correspond thus to a block of updates for each list element. Using the Var construct of GMs, and the structure of list nodes the block can be written as:

Var i **ranges over** {1, . . . , CN.S-VarObject.ListLength}
CN.S-VarObject [i].Terminal.StaticType := CN.S-Type.Terminal
endvar

Such a rule can be abbreviated as

Var v **ranges over list** CN.S-VarObject
    v.Terminal.StaticType := CN.S-Type.Terminal
endvar

where **list** l denotes the set of all elements in the list of list-node l.

$$\mathbf{list} l \triangleq \{l[i] \mid i \in \{1, \ldots, l.ListLength\}\}$$

In a similar way, quantification over the list elements can be written

for all v **in list** CN.S-VarObject **holds** ...
exists v **in list** CN.S-VarObject **such that** ...

instead of quantifying explicitly over the indexes.

At the end of section 2.2.2 we explained informally the semantics of the Parent arrow in the RecordType Montage (figure 18). The transition rule corresponding to this arrow is

Var d **ranges over list** CN.S-FieldDeclaration
    Var o **ranges over list** d,S-FieldObject
        o.Terminal.Parent := CN.S-RECORD
    endvar
endvar

The GM semantics of the static analysis defines a graph traversal algorithm. Such a traversal starts with visiting the root of the tree, and visits the nodes of the tree in a certain order. The algorithm is refering to the current visited node in the tree, this position is called the current node CN , which we used already above. For each node, the algorithm must know, which descendants have to be visited, and in which order. In our traversal algorithm, this question is answered with respect to the label of the node. The graph in the Montage of that label defines the set of visited descendants and their order as follows. All descendants which have a Montage for their symbol or one of its synonyms must be visits. The descendants are visited according to the position of their representing nodes in the graph: left before right, and upper before lower.

If the algorithm visits a node, it first visits the descendants in the given order and then it executes the static analysis transition rule as defined above. Given such a semantics with explicit traversal order, the textual parts of the static analysis can take advantage of the results of the static analysis of the descendants. This is needed for non–local aspects like references to declarations.

Another mechanism relying on the traversal order is the checking of the static semantics conditions. As we will see in section 3.3, almost all static semantics conditions use information retrieved during the static analysis of the descendants. The check of the condition is done after the static analysis of the descendants, but before the static analysis of the Montage itself. This is formalized in [29].

3 Oberon

In this section we present the specification of the programming language Oberon [39, 46]. Oberon is a direct descendent of Pascal and of Modula2 that supports object oriented programming. The main difference of Oberon (and its predecessors, and Java) with respect to C/C++ is that Oberon guarantees that the store consists of simple and composite objects, whose structure cannot be destroyed. The possible object structures are defined by declared types. Available basic types include integers, reals, booleans, and pointers, available composite types are records and arrays. In this paper we restrict us to integers and records. The other types introduce no further complexity.

Record types can be extensions of other record types. Extension [49] corresponds to inheritance in other languages. In Oberon a method is simulated by a procedure field in a record type. In [22] the language has been extended to support the usual method-call syntax. In this work, we do not show inheritance and methods, but they can be modeled with the machinery presented here, please see [25] for details. In general we abstract from features, whose use cannot be explained in the restricted available space, and who do not need additional machinery for their specification. For this reason we abstract from most control statements, expressions, modules, and procedure parameters. An idea how expressions look is given by the example in section 2. We refer to [25] for a complete specification of the dynamic semantics of Oberon.

Features that we explain in the following include the storage, allocation and updates of composite objects, local declarations, and recursive procedures. Besides of specifying the features of Oberon, we want to use this first Montages case study to show the flexibility and user friendliness of the approach. We structure this section in three parts showing typical tasks during the specification design.

- Section 3.1: Preliminary explanation of basic aspects of the language model
- Section 3.2: Specification of a complete, working sub-language
- Section 3.3: Introducing of difficult features by refinement of the existing parts

The preliminary explanation and specification of basic aspects of the language model is needed to provide intuition about the basic concepts realized by the language. Due to the modularity of Montages, this intuition can be given by the formal specification of the respective constructs, and not by simple examples as usual. Often the constructs of a language can be classified in few classes having the same abstract behavior. In [25] we classify the constructs of Oberon in variables, expressions, designators, statements, and types. The abstract class variables is modeling global and local variables, as well as variable and value parameters of procedures. After having described the properties of this abstract classes, the single constructs can be explained with reference only to the abstract classes. We explain types first as abstract class and then the concrete types. The other classes expressions, variables, statements, and designators are presented in different ways.

A complete, working sub-language is helpful for the understanding. Complete and working means, that there are no holes in the specification, and that one can thus go through examples. It is desirable to reach such a state of the specification as early as possible. Like this the reader is able to understand a small closed system, and one is motivated to understand additional constructs, rather than frustrated by the large number of constructs one has to know, in order to understand the language model.

The scenario in the last paragraph is of course only useful, if the specification approach allows for the easy introduction of additional constructs. This late introduction should first of all be possible without fundamental changes of the existing parts. If changes are needed they should be simple refinements. In [21] the specification of C is given by four specifications, each being a refinement of the previous one. Here we give in section 3.2 the specification of a complete sub-language, and show in section 3.3 how the model has to be refined in order to introduce recursive procedures with side-effects. It is remarkable, that we can do this without changing the specifications of the earlier parts.

It is clear that these techniques can be used to present and design a Montages specification of a language in many different ways. The exploitation of this design space is very easy, and requires little changes of the formal parts. In this rest we show the main language constructs, and the main specification design possibilities.

3.1 Language Model

As mentioned in this section we are going to explain some basic aspects of the language model. Most of chosen solution can be reused for other language specifications. Section 3.1.1 presents a model for the store in Oberon. Although the model is specialized for Oberon, the same solution can be taken for any strong typed language having primitive (locations) and composite (records) objects.

Section 3.1.2 presents the basic ideas how we model types. Although, as shown in [25] the modeling of inheritance and modules needs a slightly refined solution, our idea can be used to model any other object oriented type system. Unfortunately we have no place here to expose further the use of the solutions for other languages. We would like as well to stress, that this solutions are not built–in in Montages. Other solutions are of course possible, especially for languages like C/C++ byte-oriented store and type models are needed.

Section 3.1.3 shows how we deal in general with declaration mechanisms. Declarations exist for types, variables, and procedures. In section 3.1.4 we give as example the Montages of variable declarations and references, which introduce as well the concepts of static types and associated objects.

Finally in section 3.1.5 we show two statements which are used to update and allocate objects. With exception of the WHILE statement in the introduction, control statements (IF-THEN-ELSE, CASE, WHILE, REPEAT, LOOP, and FOR) are not presented. An additional basic component of Oberon would be expressions. We have shown a primitive form of expressions in section 2. Since we do not show control statements and array types in this text, expressions are not used in the following, and we refer to [25] for details.

3.1.1 Storage

The storage is modeled by a collection of abstract objects. The type of an object tells us whether it is a simple location or whether it is made of a number of other objects. For a composite object the type tells us as well how these objects can be accessed. A pointers in such a setting does not point to a concrete address, but directly to an abstract object. Therefore there is no pointer arithmetic, nor any order of the objects in the store. The objects are just elements of a universe called *Object*. In this section we explain how such a model of the store works.

If we want to allocate a new object in the store, we extend Object using the extend construct of GMs:

extend Object **with** e
    R
endextend

which extends the universe Object with a completely new element e and executes the transition rule R. Within R the local name e must be used to speak about the new element.

The set of objects is partitioned into primitive objects and composite objects.

$$Object = Primitive \cup Composite$$

Primitive objects correspond exactly to locations, we use the two terms as synonyms. A location is an object of an unstructured type, and its information is stored in a dynamic field *Value*. Composite objects are used to model instances of record (respectively class, in C++/Java terminology) and array types. Instances of such types consist typically of a composite object and a number of additional objects. These objects build typically a small tree, with the composite object as root. The descendants in the tree are accessed by a number of functions, typically binary functions whose second argument defines a descendant by a name (e.g. records) or a number(e.g. arrays). In the following we show how primitive and composite objects are used to model instances of Oberon types. We model only pointers and records in detail, since the instances of the remaining basic, procedure, and array types do not introduce additional complexity.

Pointers A pointer is modeled by a location. The possible values of a pointer are composite objects (roots of records) and the predefined element *Nil*. A newly allocated pointer is guaranteed to be initialized with Nil.

Records A record is a structured object, which consists basically of a set of named and typed fields to store instances of these types. A record is represented by a composite root object and a dynamic function

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

is used to link the root with the roots of the instances stored in the fields. The second argument of Field is the field name. In our tree metaphor a record consists thus of a root, and descendants for all its field, which are accessed using the function Field. The field-function applied to a record-root is only defined for fields of the corresponding record type. A record type defines which fields a record has, and of what type these fields are. If a field is again of record type, this field is the root of such a record. A record can thus be a tree of arbitrary complexity. The leafs of such a tree are locations and the Value fields of these locations are normally called the values of a record.

In figure 8 we show an example record of having three fields a, b, and c, where the c field is a record having two fields d, and e. All circles are objects, the record-root and the c-field are composite objects. Fields a, b, d, and e are locations, and have thus a value. The targets of the value arrows are all going in a box representing the values of the record. The Oberon type describing such a records is denoted as follows,

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

where T is an arbitrary unstructured type.

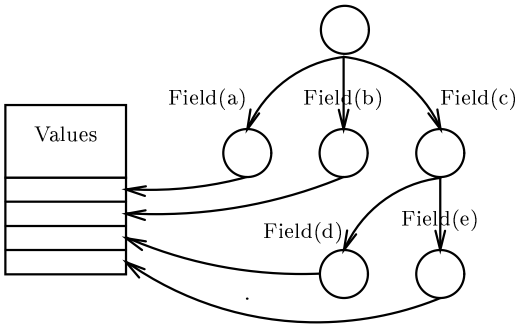


Figure 8: An example record and its values

In languages implementations, a record is linearized to a sequence of locations which hold the values of the record (please compare with the values box in figure 8). Having such a linear implementation the task to copy the values from one instance to an other is trivial. Since we abstract from a linear implementation, the task to copy all values of a record or to create such a record tree is non trivial. To solve this we propose a novel idea to model types as instance of a *class type* which provides a service for allocation and a service for copy. The structure of the type declaration can serve directly as body for these services. These *types objects* are in stark contrast to the existing static formal type systems which are currently adapted to object-oriented type systems. Our approach is not an alternative to such systems, but it allows to define a ground model for object oriented type systems, against which the static models can be verified.

3.1.2 Class Type

In this section we describe the class type abstractly. We explain the copy and create services of a type, but we defer the declaration of concrete types to section 3.2.2.

A type can be seen as an object providing services to create and copy objects. In our model, the terminal leaf of a type definition is the task representing such a service provider. We pass control to this task, if we call a service of the type; the parameters for the services are stored in three fields of the task, called *Return*, *Src* and *Dest*. The return field is used as return address of the service procedure. Thus a task calling an active type needs to put itself in the return field of the active type. Depending on the value of a dynamic nullary function

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

a type provides either the copy or the create service, if it gets control.

In *copy* mode two objects of the type are assumed to be assigned to the *Src* and *Dest* fields which are the *source* and the *destination* of the copy service. The copy service sets the value(s) of the destination to the value(s) of the source. Please remember the definition of values for structured types like records. In order to provide its service, a structured type calls typically the copy-service of other types. These calls cannot be recursive, e.g. recursive type declarations are not allowed.

We described how we model the different steps of the copy-service with GMs. In order to abstract from these technical details, we introduce a macro, which simulates a method call syntax. The macro *type*.**COPY**(*source*, *destination*) $\triangleq$

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

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

In *create* mode, a new instance of the type is created. In the case of structured types (like records or arrays) such an instance is a tree of objects; in the case of unstructured types it a single object. The root of the instance is accessible as the *Dest* field of the type. The standard method to call a type in create mode is given by a macro

type.**CREATE()** ; R $\triangleq$

if not CT.Allocated **then**
  Mode := create
  type.Return := CT
  CT := *type*
  CT.Allocated := true
else
  CT.Allocated := false
  R
endif
where NewObject $\triangleq$ *type*.Dest

In this macro we define the local macro **NewObject** which can be used in R to access the new object *type*.Dest. The semicolon ";" in the macro is used to indicate, that transition rule R is executed after the allocation. The sequential execution of the steps is forced in the macro-definition using a boolean field Allocated.

The first paragraph on pointer types in section 3.2.2 is a simple example for the copy and create services of the pointer instance of class type. We propose to the reader to examine it already now, if he wishes to see a concrete example. A more general description of the class type and the modeling of object oriented type systems must be defered to future work.

3.1.3 Declarations

One of the main mechanism in programming language is the declaration of named structures, and their use in the algorithms. Such structures include parts of the store, known as *variables*, code fragments, known as *procedures*, and properties of variables and procedures, so called *types*. In modern programming languages all three types of structures are created by declarations. A declaration consists typically of an identifier and a characterization. Types are characterized by a construction from other types (e.g. record, pointer types), variables are characterized by types, and procedures are characterized by the type of their parameters, a code fragment using these parameters, and additional declarations. The structures are declared once but can be used several times in the code. In order to use the structures one references them using their identifier. In our approach we model that situation as follows.

The structures are represented directly by the identifier in their declaration. We design the context–free grammar such, that the identifier in a variable declaration is labeled with non-terminal VarObject. The identifier in a procedure (respectively type) declaration is labeled with ProcObject (respectively TypeObject). All these nonterminals are synonyms of Ident:

VarObject	=	Ident
ProcObject	=	Ident
TypeObject	=	Ident

The micro syntax of an Ident is accessed as the attribute

$$Name : Ident \rightarrow String$$

which holds the scanned value of the identifier. As noted, the references to the declared structures are identifiers as well. But a variable, procedure, or type reference is additionally labeled with Simple, TypeIdent, or Call, respectively. A reference identifier is declared by a declaration identifier having the same micro syntax. The declaration is a data-flow attribute Decl:

$$Decl : Simple \cup TypeIdent \cup Call \rightarrow VarObject \cup TypeObject \cup ProcObject$$

This attribute must be set by the static analysis. This cannot be specified graphically, since we do not know, where in the code a declaration and its references are. But we know, that during the tree traversal, we visit first the declarations and then the references. When we visit a declaration, we note in a table its identifier node and its name. If we later visit a reference we can look up in this table the node of its declaration. We use three different tables for variables, procedures and types:

$$\begin{aligned} VarTABLE : &\quad String \rightarrow VarObject \\ ProcTABLE : &\quad String \rightarrow ProcObject \\ TypeTABLE : &\quad String \rightarrow TypeObject \end{aligned}$$

In the next section we show in detail the semantics of variable declarations and use which are a good example for the above described mechanisms. Details of the declaration and use of procedures and types is included in the corresponding sections 3.2.2 and 3.3.

3.1.4 Variables

In this section we give the Montages for variable declarations and simple references. These specifications exemplify not only the declaration mechanisms described in the last section, but they introduce as well the concept of static type, they use for the first time the create service of types, and they show how the variable are linked to their associated objects in the store.

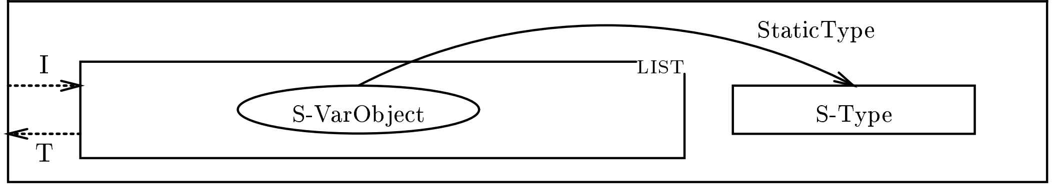
We mentioned in the last section that a variable is characterized by a type. This type is stored as attribute *StaticType* of a VarObject. In the upper Montage of figure 9 we see how a variable declaration looks, and how the StaticType attribute is defined. The static type of a variable defines which values can be stored in a variable, and how these values can be read. In our approach we model this as follows. A variable object is a place where we can store exactly one object of the store universe Object. We call this object the associated object of the variable and the field in which we store the object is called AssocOBJ. The strong typing of Oberon guarantees that a variable of type T is associated with the root of an instance of T. The type of the variable defines what kind of object is associated with a variable. Some kinds, for instance records, are useful to store many values, others like pointers are useful to store single values. Please remember that values include other objects. In section 3.2.2 we will see how the different types are declared, and which kind of objects correspond to them. For the moment we do not need to know the details of the existing types.

In the middle Montage of figure 9 static analysis and dynamic semantics of a variable object is defined. The static semantics condition is given in the Montage for declaration sequences, it says that in a declaration sequence the name is not used twice. The static analysis of a variable object updates the variable table. The dynamic semantics allocates a new object in the store, and sets the AssocOBJ field to that new object. This rather complex process is very useful, since it will allow to use the same model for local variables.

The lower Montage of figure 9 specifies all aspects of simple variable references. Such a reference is a special case of designators, which are defined in section 3.2.3, where we will see how to construct from a references to an object, references to the value or the descendants of the object. The static semantics condition in the Simple Montage guarantees that there is a corresponding entry in the declaration table. This entry is denoted in the static analysis rule by the local macro **D**. The rule sets the Decl field to **D**, and the static type to **D**.StaticType. Finally the dynamic semantics rule sets the AssocOBJ field of a Simple to the associated object of its declaration.

Please note that this model is enough general to model even local variables, variable and value parameters. The use for local varibles is shown in section Procedures, and for variable and value parameters we refer to [25].

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



VarObject **=** **Ident**

VarTABLE(CN.Name) := CN

if VarObject(CT) **then**
CT.StaticType**.CREATE()** ;
CT.AssocOBJ:= **NewObject**
CT := CT.NT
endif

Simple **=** **Ident**

CN.Decl := **D**
CN.StaticType := **D**.StaticType
where D $\triangleq$ VarTABLE(CN.Name)

condition VarTABLE(CN.Name) $\neq$ undef

if Simple(CT) **then**
CT.AssocOBJ:= CT.Decl.AssocOBJ
CT := CT.NT
endif

Figure 9: The semantics of a variable declaration, a variable object and a simple designator

3.1.5 Statements

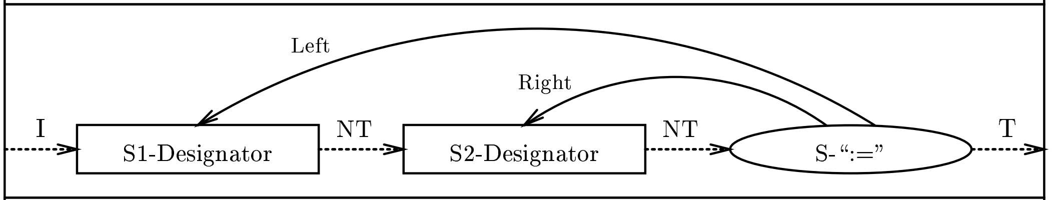
StatementSequence ::= Statement {“;”Statement}

The statements in a statement sequence are sequentially linked through the next task function in their syntactical order. This is done implicitely through the ListNode definitions. The different statements that can be sequenced are:

Statement ::= Assignment | NewStatement | ProcedureCall

The assignement is used to assign the value of the right designator to the value of the left designator. New objects are allocated using the predefined procedure NEW. The task of a NEW-node n is to allocate a new object and to assign it to the the *Value* field of the associated object of the pointer *n.Pointer*. The semantics of assignment and new statement is given by Montage 10 respectively Montage 11.

Assignment ::= Designator “:=” Designator

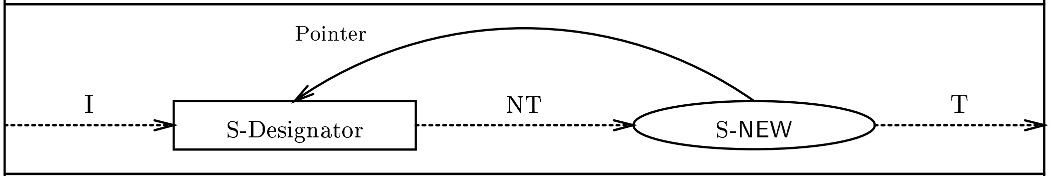


condition CN.S1-Designator.Terminal.StaticType = CN.S2-Designator.Terminal.StaticType

```
if “:=”(CT) then
  CT.Left.StaticType.COPY(CT.Right.AssocOBJ, CT.Left.AssocOBJ)
endif
```

Figure 10: The semantics of an assignment

NewStatement ::= NEW“(”Designator“)”



condition POINTER(CN.S-Designator.Terminal.StaticType)

```
if NEW(CT) then
  CT.Pointer.StaticType.ReferencedType.CREATE() ;
    CT.Pointer.AssocOBJ.Value := NewObj
    CT := CT.NT
endif
```

Figure 11: The semantics of a new statement

The new procedure is a special case of procedure call. We treat the general case together with recursion in section 3.3.

3.2 A Working Subset

After having specified basic aspects of Oberon in the last section, we extend now the given parts to a complete, working sub-language of Oberon. This language contains all constructs concerned with allocation, reference, and update of storage. The language allows for all possible pointer manipulations in Oberon. The aspect of pointer manipulation is seldom treated in language specifications, since it is not easily treated with traditional specification approaches.

In section 3.2.1 we present the start symbol Program of our sub-language. This symbol does not really exist in Oberon, and is used to abstract from the module concept. The following section 3.2.2 shows how types are declared and referenced in Oberon. Finally section 3.2.3 presents the designators which are used to access objects in the store.

3.2.1 Programs

In Oberon code is executed, if a module is initialized or a command is called. In this text we treat neither modules nor commands, and therefore we introduce a fictive program construct. This construct is specified by the Montage in figure 12. The non-terminal Program is the start symbol of our grammar. The textual part CT := CT.S-DclSequence.Initial of the static analysis is typical for the Montage of a start symbol. After the descendants have been visited, this textual part sets the current task to the first task that has to be executed, and the CN is set to CN.Up, which is undefined for the root node. Thus the static analysis rules will fire no more, and the dynamic semantics rules start to fire. This immediate change from static analysis to dynamic semantics can of course be avoided by suppressing the textual rule in figure 12. Then the rules terminate after the static analysis, and the current task must be set externally to the first task.

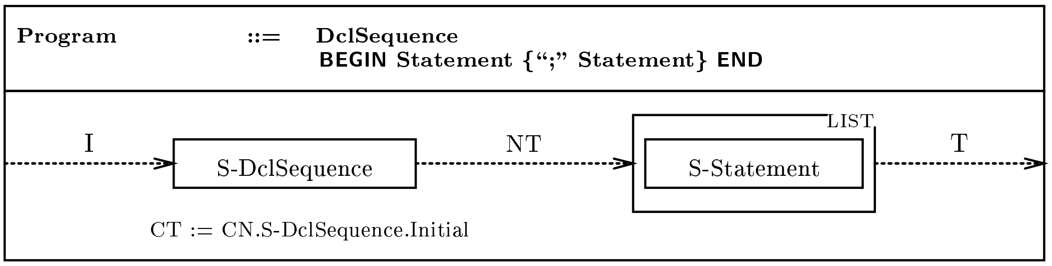
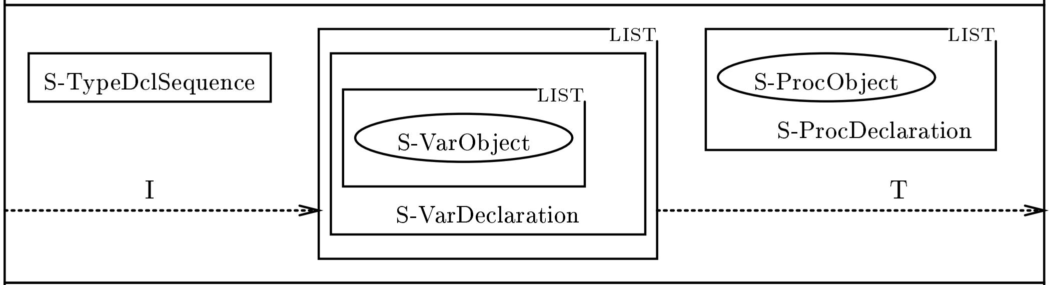


Figure 12: The semantics of an Oberon program

The declaration sequence in the program is the place where types, variables, and procedures are declared. Type declarations are visited twice, since Oberon allows pointers to not yet defined record types. This needs two scans. The semantics of the two visits will be different, depending on the tags FirstVisit and SecondVisit. Tags can be seen as global switches which can be checked by the static analysis of the descendants.

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



condition **for all** d1, d2 **in list** CN.S-VarDeclaration **holds**
  for all v1 **in list** d1.S-VarObject **holds**
    for all v2 **in list** d2.S-VarObject **holds**
      v1 $\neq$ v2 **implies** v1.Name $\neq$ v2.Name
and
for all d1, d2 **in list** CN.S-ProcDeclaration **holds**
  d1 $\neq$ d2 **implies** d1.S-ProcObject.Name $\neq$ d2.S-ProcObject.Name

Figure 13: The semantics of a declaration sequence

3.2.2 Type Declarations

In this section we present type declarations and some concrete types, like pointer and record types. A type declaration, as specified by the Montage in figure 14 works exactly like a variable declaration, except that we use the declaration table TypeTABLE instead of VarTABLE.

TypeDeclartion ::= **TypeObject "=" Type**
TypeObject = **Ident**

TypeTABLE(CN.S-TypeObject.Name) := CN.S-Type.Terminal

Figure 14: The semantics of a type declaration

Up to now we abstracted which types do exist in our language. All types are instances of the class type, which we presented in section 3.1.2. In this section we present concrete basic and constructed types together with the semantics of their copy and create services. The types of our sub-language are divided in type identifiers, pointer types and record types.

Type	=	TypeIdent | RecordType | PointerType
TypeIdent	=	Ident

Type identifiers Type identifiers are specified in Montage 15. They serve as references to user defined types or to predefined basic types. The predefined basic types SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, BOOLEAN, and CHAR provide on our abstraction level equivalent the same services. Therefore we restrict us to one example, the type INTEGER, which can be seen as instance of a sub-class basic-type of the class type.

TypeIdent = **Ident**

```
if CN.Name = INTEGER then
  IntegerType(CN) := true
else
  CN.Initial := D
  CN.Terminal := D
endif
where D ≜ TypeTABLE(CN.Name)
```

condition CN.Name = INTEGER **or** TypeTABLE(CN.Name) ≠ undef

```
if IntegerType(CT) then
  if Mode = create then
    extend Primitive with i
      CT.Dest := i
    endextend
  endif
  CT := CT.Return
endif
```

```
if IntegerType(CT) then
  if Mode = copy then
    CT.Dest.Value := CT.Src.Value
  endif
  CT := CT.Return
endif
```

Figure 15: The semantics of a type identifier

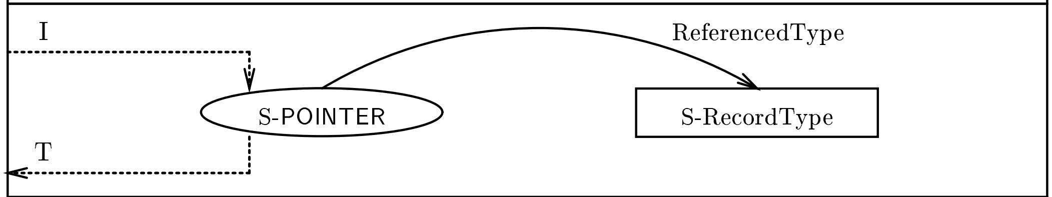
If a type identifier references not a predefined, but a user defined type T, we conceptually "replace" the reference with T. Since only the initial and terminal leaf are of importance for the rest of the model, we can "replace" the reference by setting its Terminal and Initial attribute to the terminal/initial leaf of T. In the static analysis rule of Montage 15 we define how this leaf is assigned to the initial and terminal of the type identifier. The dynamic semantics rule defines the copy and create services of an integer type. The integer types are characterized by the static semantics by putting them in the universe IntegerType.

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

PointerType = PointerToIdent | PointerToRecord

Pointer types to record types, as defined in figure 16, can check their static semantics and perform their static analysis directly. The ReferencedType denotes the type to which the pointer "points". Since this type is syntactically guaranteed to be a record type, there is no static semantics condition. The dynamic semantics of a pointer type applies independently whether the type was generated by PointerToRecord or PointerToIdent. The rule has a part which defines the create service and which defines the copy service. The create service allocates a new location and the value of the new location is initialized with Nil, the copy service copies the value field of the source to the value field of the destination.

PointerToRecord ::= **POINTER TO RecordType**




```
if POINTER(CT) then
  if Mode = create then
    extend Primitive with o
      CT.Dest := o
      o.Value := Nil
    endextend
  endif
  CT := CT.Return
endif
```

```
if POINTER(CT) then
  if Mode = copy then
    CT.Dest.Value := CT.Src.Value
  endif
  CT := CT.Return
endif
```

Figure 16: The semantics of a pointer type

The semantics of PointerToIdent has some subtle points, which are not defined in the language definition [47]. Nevertheless we show one specific solution, which is according to [47] not wrong.

The lower Montage in figure 17 shows the semantics of a pointer to identifier. The static semantics condition is that the name of the Ident is not INTEGER, since pointers to integer variables are not supported by the languages. In our sub-languages pointers must point to record types, a condition which is checked by the TypeDeclSequence Montage (see below).

Since we are allowed to reference not yet declared types in a pointer to ident, the static analysis can only be done, after all types have been declared. Thus the main parts of static analysis and consequently as well of static semantics of a pointer to ident must be done by the type declaration sequence (see upper Montage in figure typedclsequence). The type declaration sequence must thus now all pointer types for which it has to “fix” the static analysis and semantics. For this purpose the static analysis of PointerToIdent includes these pointers in a universe PointerToIdentSet.

$$PointerToIdentSet(CN) := true$$

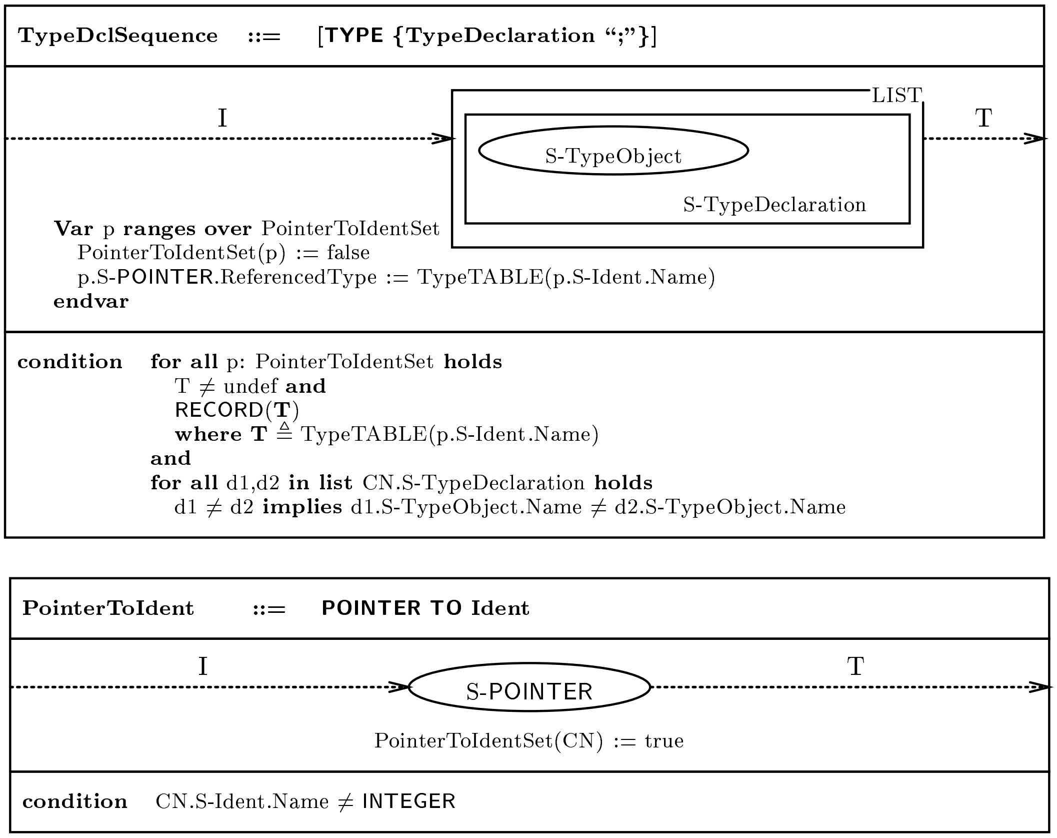


Figure 17: The semantics of a type declaration sequence and a pointer to identifier type

As mentioned, part of the work of the type declaration sequence (upper Montage in figure 17) is to fix static analysis and to check static semantics for all pointer types p of PointerToIdentSet. The static semantics condition checks whether the referenced type of p is declared, and whether the declared type is a record type. The static analysis sets the ReferencedType attribute of p to the declared referenced type. At least p is excluded from the universe PointerToIdentSet

$$PointerToIdentSet(p) := false$$

Independent from pointer types, the second part of the static semantics condition of a type declaration sequence guarantees, that the names of all declared types are different.

Record types In the following the record type constructor is explained. Its generality is the reason why the solution with the class type as abstract service provider has been chosen.

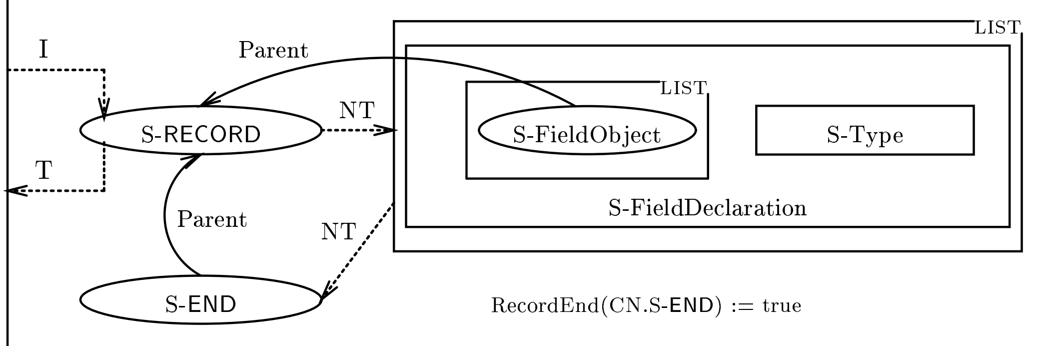
Other solutions like higher order functions do not scope to the complexity of these record types. The class type on the other hand showed to be suited for the record types of full Oberon, featuring inheritance and complex encapsulation mechanisms. It is worth noting, that these mechanisms can be used directly for the specification of method overloading, as provided by Oberon2 and C++/Java.

A record type definition consists of a list of field declarations. These declarations define *fields* of the record type. The semantics a record type and a field declaration are given by the Montages in figure 18. The static semantics condition of RecordType defines that there are not two fields with the same name. The formula is rather complex, since fields are not declared in one list, but as a list of lists of fields. The static analysis of a record type sets the attribute Parent of each field object to the RECORD-token, and defines the control-flow such that sequentially the RECORD-token, then all field objects and at the end the END-token will get the control. The static analysis of FieldDeclaration sets the attribute StaticType of each field object to the given type.

The dynamic semantics rules define the create and copy service of record types. As we see the services are executed by the different tokens in create and copy mode. The create service is called by passing control to the RECORD-token while being in create mode. The RECORD-token allocates a record r, then each field object f_i creates an object o_i using the the create mode of its type, and then the corresponding field f_i of r is set to o_i. After all field objects had the control, the control is passed to the END-token that passes the control to the return address of the record type, which terminates the create service. The copy service is correspondingly called by passing control to the RECORD-token while being in copy mode. The RECORD-token passes control directly to the field objects which copy the values of the corresponding fields by calling the copy services of their types.

In our model arbitrary complex type constructions are possible. In real implementations the complexity of such constructions is typically restricted, since most of the efficient compiler techniques assume bound-resources, and efficient implementations on these resources allow typically not for arbitrary complex types. Although the restrictions are normally not considered as part of the language definition, they can and should be formalized as static semantics conditions, if a specific implementation is specified. Here we do not fix such restrictions, since we are not speaking about a specific implementation of the language.

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



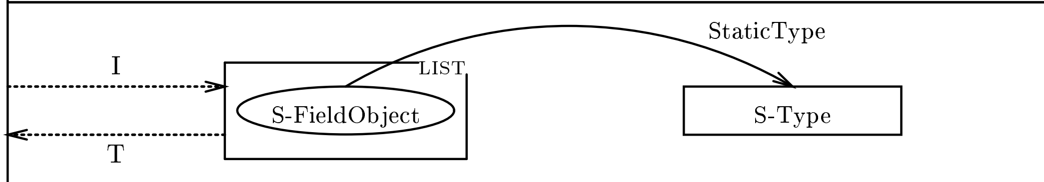
condition **for all** d1,d2 **in list** CN.S-FieldDeclaration **holds**
  for all f1 **in list** d1.S-FieldObject **holds**
    for all f2 **in list** d2.S-FieldObject **holds**
      f1 $\neq$ f2 **implies** f1.Name $\neq$ f2.Name

```
if RECORD(CT) and Mode = create then
   extend Composite with o
      CT.Dest := o
      CT := CT.NT
   endextend
endif
```

```
if RECORD(CT) and Mode = copy then
   CT := CT.NT
endif
```

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

FieldDeclaration ::= **FieldObject** {"**,**" **FieldObject**} "**;**" **Type**
FieldObject = **Ident**




```
if FieldObject(CT) and Mode = create then
   CT.StaticType.CREATE() ;
      Field(CT.Parent.Dest, CT.Name) := NewObject
      CT := CT.NT
endif
```

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

Figure 18: The semantics of a record type and a field declaration

3.2.3 Designators

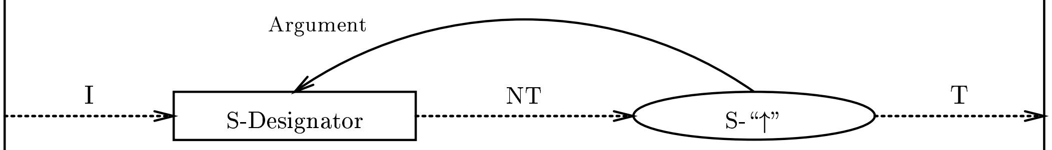
Designators are used to reference objects in the store. From section 3.1.4 we know already the simple designator, which references the associated object of a variable. Like the complex designators specified below, the simple designator is evaluated by passing control to it, and during this evaluation it stores its referenced, or associated object in the AssocOBJ-field. Designators can thus be seen as instances of a class designator, that has a public field AssocOBJ, and that provides one service, which updates AssocOBJto the currently referenced object. To call this service one passes just control to the designator.

All designators are synonyms of the non-terminal Designator.

Designator	=	Simple | Qualified | Dereferenced

The dereferenced designator, specified by the Montage in figure 19, is used to construct from a reference to a pointer p, a referenced to the pointed object p.Value. If the pointed object is Nil, a runtime error is raised. The static semantics condition of a dereferenced designator is of course that the type of the argument is a pointer type.

Dereferenced ::= **Designator"↑"**



CN.S-"↑".StaticType := CN.S-Designator.Terminal.StaticType.ReferencedType

condition POINTER(CN.S-Designator.Terminal.StaticType)

```
if "↑"(CT) then
  if CT.Argument.AssocOBJ.Value ≠ Nil then
    CT.AssocOBJ:= CT.Argument.AssocOBJ.Value
    CT := CT.NT
  else
    CT := Trap
endif
```

Figure 19: The semantics of a dereferenced designator

The qualified designator (lower Montage in figure 20) is used to access fields of records. The argument designator may reference either a record or a pointer. Which of the two cases is present can be decided statically. The type of the argument designator is denoted by macro ArgType.

ArgType $\triangleq$ CN.S-Designator.Terminal.StaticType

If ArgType is a record type, the field selector selects directly the field of the argument, thus we include it in the universe DirectFieldSelector. If ArgType is a pointer type, the field selector selects after dereferencing the pointer implicitly. We include such field selectors in the universe IndirectFieldSelector.

The macro SetStaticType to is used to set the static type of the qualified designator to the type of the corresponding field. The first argument of the macro is the record type, which is either directly ArgType, or for indirect field selectors it is the referenced type of ArgType. The second argument is the name of the field. The definition of SetStaticType is a range over the list of field declarations, and over the list of field objects in each field declaration; during this range, it sets the type of the FieldSelector to the type of the field with the right name. The static semantics of RecordType guarantees, that there is at most one such field, and the static semantics of Qualified guarantees that there is such a field.

SetStaticTypeTo(*type*, *name*) $\triangleq$

Var d **ranges over list** *type*.Up.S-FieldDeclaration
  Var o **ranges over list** d.S-FieldObject
    if o.Name = *name* **then**
      CN.S-FieldSelector.StaticType := f.StaticType
    endif
  endvar
endvar

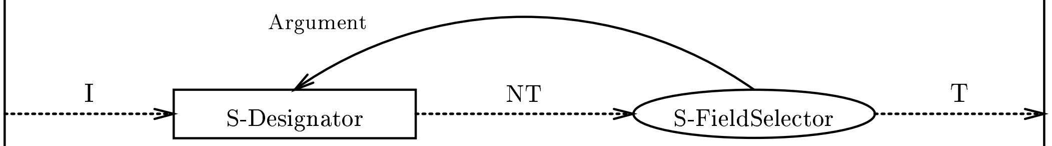
The static semantics condition of Qualified is that the respective field in the record type is defined. To look for the fields, we have to scan the list of lists of field objects. The macro FieldDefined does this, with the record type as first argument and the field name as second argument. Field defined yields true, if there is a field with that name in the record type.

FieldDefined(*type*, *name*) $\triangleq$

exists d **in list** *type*.Up.S-FieldDeclaration **such that**
  exists o **in list** d.S-FieldObject **such that**
    o.Name = *name*

The dynamic semantics rules for direct and indirect field selectors are different. A direct field selector just sets its associated object to the corresponding field of the associated object of the argument. In the case of an indirect field selector, there is the possibility that the associated object of the argument is a pointer to Nil. In this case, the implicit dereferencing of that pointer leads to a run-time error, and control is passed to Trap. Else, the argument is dereferenced and the associated object is set to the corresponding record-field.

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



if RECORD(**ArgType**) **then**
  SetStaticTypeTo(**ArgType**, CN.S-FieldSelector.Name)
  DirectFieldSelector(CN.S-FieldSelector) := true
elseif POINTER(**ArgType**) **then**
  SetStaticTypeTo(**ArgType**.ReferencedType, CN.S-FieldSelector.Name)
  IndirectFieldSelector(CN.S-FieldSelector) := true
endif

condition (RECORD(**ArgType**) **and**
  FieldDefined(**ArgType**, CN.S-FieldSelector.Name))
or (POINTER(**ArgType**) **and**
  FieldDefined(**ArgType**.ReferencedType, CN.S-FieldSelector.Name))

if DirectFieldSelector(CT) **then**
  CT.AssocOBJ:=
    Field(CT.Argument.AssocOBJ,
      CT.Name)
  CT:= CT.NT
endif

if IndirectFieldSelector(CT) **then**
  if CT.Argument.AssocOBJ.Value $\neq$ Nil **then**
    CT.AssocOBJ:=
      Field(CT.Argument.AssocOBJ.Value, CT.Name)
    CT:= CT.NT
  else
    CT := Trap
  endif
endif

Figure 20: The semantics of a qualified designator

3.3 Procedures

In its simplest form, a procedure is just a named sequence of statements. The statement sequence of the procedure is its body. A procedure call is a statement which executes the body of the procedure. Usually procedures encompass as well

- the concept of a result: a procedure delivering a result can be us part of expressions;
- the concept of parameters;
- the concept of local types, variables and procedures.

In our example language, we skip the concept of a result, since we skipped expressions all together. We abstract as well from parameters, since they are only partially useful without expressions, and since they introduce no additional complexity to the model.

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

Syntactically local declaration look exactly like the declaration sequence introduced in figure 13. A code object (e.g. variable, type, procedure) which is locally declared in a procedure can only be referenced within that procedure, else there would be no encapsulation. Externally declared code objects can as well be referenced in a procedure, if their name has not been reused in a local declaration.

The section of the code in which a code object can be referenced is called its scope. The scope of a variable (type, procedure) declared in a program is the code following its declaration. If the declaration is local to a procedure, the scope ends at the end of the procedure. As already mentioned such a scope has a "hole", if the name of the variable (type, procedure) is reused for the declaration of a variable (type, procedure). The reuse of names is not allowed within the same declaration sequence. But procedure declaration within the same declaration section have an own local declaration sequence allowing to reuse all names. The "hole" in the scope starts where the name is reused in a declaration and ends at the end of procedure containing that declaration. As we see this rules contain many subtle points.

In our setting we model the concept of scope as a natural number. The scope of the program is 1. The scope of a procedure local to a program is 2. The scope of a procedure local to a procedure with scope n is $n+1$. For each scope we introduce new declaration tables, thus we parameterize the declaration table functions with the scope.

$$\begin{aligned} VarTable: &\quad Nat \times String \rightarrow VarObject \\ TypeTable: &\quad Nat \times String \rightarrow TypeObject \\ ProcTable: &\quad Nat \times String \rightarrow ProcObject \end{aligned}$$

During the static analysis, we guarantee that a dynamic function Scope

$$Scope : Nat$$

denotes the scope of the currently analyzed code. In all up to now defined rules, the declaration tables are used on the current scope. Thus they are still valid if we refine the functions VarTABLE, TypeTABLE, and ProcTABLE to the following macros:

$$
\begin{aligned}
\mathit{VarTABLE}(name) &\triangleq \mathit{VarTable(Scope,\ name)}\\
\mathit{TypeTABLE}(name) &\triangleq \mathit{TypeTable(Scope,\ name)}\\
\mathit{ProcTABLE}(name) &\triangleq \mathit{ProcTable(Scope,\ name)}
\end{aligned}
$$

If we enter a new scope, the old declaration tables are copied to the new ones. Like this the old objects are visible until they are overwritten in the table. This matches exactly the informal scoping rules.

We can now explain the static analysis and semantics of the Montages procedure declaration, procedure object, and procedure call in figure 21. The static analysis of a procedure declaration analyses first its procedure object (see middle part of figure 21). The static analysis of that object, increments the scope, and copies the declaration tables of the old scope. In addition it enters itself in the procedure declaration table of the old scope and the new scope. The static analysis of the procedure declaration continues by analyzing the local declaration sequence and the statement sequence. After this the increased scope is decremented to its old value.

The semantics of a procedure call is given as lower Montage of figure 21. The static part correspond more or less to a simple designator, the call gets its declaration from the procedure declaration table, and the static semantics guarantees that there is an entry. In order to understand the dynamics, we have to look at recursion first.

If code is in execution, two kind of results are produced by the already executed tasks and can be used by the tasks that are still going to be executed:

- The allocated or copied associated object of a variable V is stored in AssocOBJ(V).
- The calculated associated object of a designator D is stored in AssocOBJ(terminal(D))

If the same task is executed twice (e.g. if its part of a loop) the results stored by *AssocOBJ* are overwritten. All kind of loop statements rely on this behavior.

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

To resolve the above problem we use the same technique as for the scopes. We parameterize the attribute holding the associated object with the recursion level.

$$\mathit{AssoccObj} : \mathit{Nat} \times (\mathit{VarObject} \cup \mathit{Simple} \cup \text{“}\uparrow'' \cup \mathit{FieldSelector}) \rightarrow \mathit{Object}$$

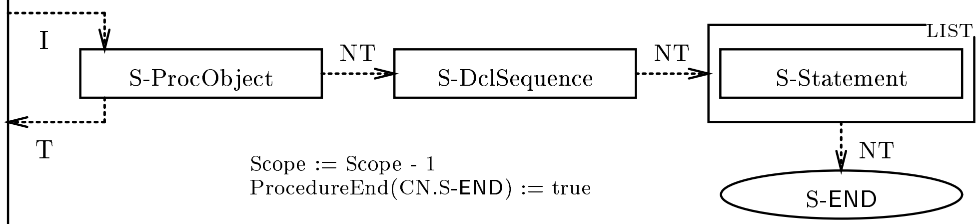
We introduce an nullary dynamic function *RecursionLevel* (or short *RL*) that indicates always the current recursion level.

$$\mathit{RL} : \mathit{Nat}$$

Since we access the associated objects up to now only on the current recursion level, we refine the function AssocOBJ to a macro with the following definition:

$$\mathit{AssocOBJ}(n) \triangleq \mathit{AssoccObj(RL,\ n)}$$

ProcDeclaration ::= **PROCEDURE ProcObject"();" DclSequence BEGIN Statement {";" Statement} END Ident**




```
Scope := Scope - 1
ProcedureEnd(CN.S-END) := true
```

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

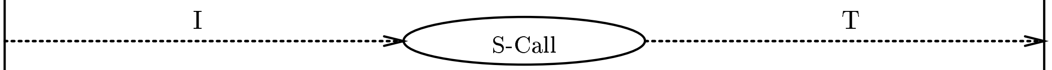
```
if ProcedureEnd(CT) then
   RL := RL -1
   CT := Caller(RL).NT
endif
```

ProcObject = **Ident**

```
ProcTABLE(CN.Name) := CN
Var n ranges over String
   VarTable(Scope + 1, n) := VarTABLE(n)
   TypeTable(Scope + 1, n) := TypeTABLE(n)
   if n = CN.Name then
      ProcTable(CScopeL + 1, n) := CN
   else
      ProcTable(CScopeL + 1, n) := ProcTABLE(n)
   endif
endvar
Scope := Scope + 1
```

```
if ProcObject(CT) then
   CT := CT.NT
endif
```

ProcedureCall ::= **Call"()"**
Call = **Ident**




```
CN.Decl := ProcTABLE(CN.Name)
```

condition ProcTABLE(CN.Name) $\neq$ undef

```
if Call(CT) then
   Var o ranges over VarObject ∪ Simple ∪ "↑" ∪ FieldSelector
      AssoccObj(RL + 1, o) := AssoccObj(RL, o)
   endvar
   Caller(RL + 1) := CT
   RL := RL + 1
   CT := CT.Decl
endif
```

Figure 21: The semantics of a procedure declaration, a procedure object and a procedure call

Like this all up to now defined Montages are as well valid in the model including recursion.

The dynamic semantics rule in the lower part of figure 21 does now the following. If a procedure is called, the recursion level is incremented with one. The value of AssocOBJ in the old recursion level is copied to the new one, in order to use non local variables. During executions the definition of AssocOBJ on the new level may change.

After execution of the procedure body the control is given back to the caller and the recursion level decremented with one (See dynamic semantics rule of ProcedureEnd in the upper part of figure 21). Thus the informations calculated before the call are still available.

The only problem we have not solved is how a procedure can pass back the control to the task, by which it has been called. Since procedures can be called recursively the solution with the return field, as chosen for types, cannot be applied here. We introduce a unary dynamic function Caller: Nat $\rightarrow$ Call, which is set in the dynamic semantics rule of a call. As we see, the call stores itself as the caller of the new recursion level. The procedure is executed on the new recursion level, and at the end control is passed back to the next task of the caller (see dynamic semantics rule of ProcedureEnd).

4 Conclusions and Further Work

Although last years have been a time of awakening and rapid growth in software technology, it seems that language description formalisms are still causing certain dissatisfactions. This paper illustrated how Montages can address the difficult problem of specifying a modern programming language, keeping into account pragmatical aspects of language design. Thus, among the major goals there was to have a formal method which was equally good understood by language designers, compiler constructors, and programmers.

GMs has been already shown ([20]) as one of the best formalism to give dynamic semantics to realistic programming languages in a compact and understandable way. The classical use of GMs abstracts from the static semantics and assumes the result of a static analysis. The main criticisms against such an approach is that the static analysis is not formalized. Montages solves this problem using control– and data–flow graphs and at the same time allows to give a very compact definition of static semantics.

Montages are attractive for language designers since the appropriateness for the programming language life cycle: requirement, design, specification, prototyping, testing, implementation, and maintenance can fruitfully use the same formalism. The designer can keep track of her/his decisions and can convey the intentions to the implementors relying on a formal setting, which is at the same time highly readable and comprehensible. Modularity and syntax–driven hierarchical decomposition allow to cope with the overall complexity of the design. Behavior localization – each non synonym production is a behavior locality – enhances maintenance.

We experienced how scaling up both basic GMs and Montages for large case studies such as the specification of SQL [16] and Oberon [25] showed some important advantages of Montages with respect to basic GMs. The specification process starting from the static semantics and analysis allows one to have a better understanding of the dynamic semantics. Less time is required since the design is driven by the structure of the Montages, which have very neat boundaries. Each Montage represents a certain quantum of behavior which can be reused in the definition of other languages in a “off-the-shelf” manner.

The Oberon case study presented in section 3.3 showed how the design of a complex language requires continuous refinements and backtracking: rarely the final decisions we have taken were the ones we sketched at the beginning. Nevertheless, it was possible to keep the overall design under control and to allow cooperative design between different designers.

At the moment at the University of L'Aquila a Montages tool, called *Darkroom*, is under development; we integrate in it a graphical front–end, a parser generator, and a GMs interpreter: providing a Montage specification of a language the tool generate the parser, the checker and an interpreter.

GMs have attracted interests and criticisms since they were invented. As pointed out in [5] GMs are deliberately not endowed with a verification system. The ease of abstraction makes GMs very suitable for different application domains and for each of them it makes sense to have a different verification system with its own assumptions. It is possible to translate a GM model in other formal frameworks in which one can do some specific verifications. Actually, in [27, 26] we translated GMs in Slang/Specware [41] respectively Natural Semantics/Centaur [23, 12]. Doing so we are able to do proofs in those systems. On the other hand one is able to do complex mathematical proofs directly in the GMs framework, as shown in [5, 11, 9].

The solution for the static aspects of our approach has some similarities with attribute grammars [24], which have been introduced to include semantics in context–free grammars. Although in certain cases they may be executed very efficiently, we preferred not to use them for the following reasons: using GMs we have the same formalism for all parts of the specification; and as shown in [31] attribute grammars tend to be very long if applied to real programming languages.

Although it is far more natural to specify state based programming languages in a state based formalism like GMs, there are functional approaches that scope surprisingly well with the problem. Natural Semantics [23] is one of the most prominent of these approaches and we used it for the dynamic semantics of Oberon [26]. Although we succeeded due to the excellent tool support by Centaur [12], the result was much longer and more complex then the GM counterpart given in [25], since one has to carry around all the state information in the functional case.

Using GMs for dynamic semantics, the work in [35] defines a framework comparable to ours. Although it has different aims, namely efficient execution. For the static part, it proposed occurrence algebras which integrate term algebras and context free grammars by providing terms for all nodes of all possible derivation trees. This allows such an approach to define all static aspects of the language in a functional algebraic system, which is supported by the MAX system [34].

Montages are being applied at International Computer Science Institute in Berkeley for giving the formal semantics to the object–oriented language Sather [2].

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison–Wesley, 1985.

[2] M. Anlauff. The Semantics of the object–oriented programming language Sather. Technical report, International Computer Science Institute, Berkeley, 1996. in preparation.

[3] 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. implementation on the Centaur system [12] is accessible through INRIA, Sophia-Antipolis.

[4] E. Börger. Annotated Bibliography on Evolving Algebras. In E. Börger, editor, *Specification and Validation Methods*, pages 37 – 51. Oxford University Press, 1995.

[5] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering. In *SOFTSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236 – 271. Springer Verlag, 1995.

[6] E. Börger and I. Durdanović. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, 39(1):52 – 92, 1996.

[7] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: The Primary Model. In *IFIP 13th World Computer Congress, Volume I: Technology/Foundations*, pages 489 – 508. Elsevier, Amsterdam, 1994.

[8] E. Börger, U. Gläser, and W. Mueller. Formal Definition of an Abstract VHDL'93 Simulator by EA–machines. In C. Delgado Kloos and Peter T. Breuer, editors, *Semantics of VHDL*, volume 307 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer, 1995.

[9] E. Börger and S. Mazzanti. A Correctness Proof for Pipelining in RISC Architectures. Technical Report 22, DIMACS: Rutgers Univ., Princeton Univ., AT&T Bell Lab. and Bellcore, 1996.

[10] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 1994.

[11] E. Börger and D. Rosenzweig. *The WAM - Definition and Compiler Correctness*, chapter 2, pages 20 – 90. Series in Computer Science and Artificial Intelligence. Elsevier Science B.V.North Holland, 1995.

[12] 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.

[13] J.T. Boyland. Conditional Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73 – 108, 1996.

[14] N. Chomsky. Three Models for the Description of Language. *IRE Trans.on Information Theory*, IT–2:113 – 124, 3 1956.

[15] G. Del Castillo, Durdanović I., and U. Glässer. *An Evolving Algebra Abstract Machine*, volume 1092 of *LNCS*, pages 191 – 214. Springer Verlag, 1993. more infos on http://www.uni-paderborn.de/fachbereich/AG/agklbue/staff/igor/ASM/asm/.

[16] B. DiFranco. Semantica Statica e Dinamica di SQL mediante i Montaggi. Master's thesis, Universitá di L'Aquila, 1996. (in italian), in preparation.

[17] P et al.. Glavan. Evolving algebra model of programming language semantics. In *Technology and Foundations Information Processing '94*, volume A–51, pages 416 – 422. IFIP Transactions A (Computer Science and Technology), 1994.

[18] Y. Gurevich. Logic and the Challange of Computer Science. *Theory and Practice of Software Engineering*, pages 1 – 57, 1988.

[19] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[20] Y. Gurevich and J.K. Huggins. *The Semantics of the C Programming Language*, volume 702 of *LNCS*, pages 274 – 308. Springer Verlag, 1993. An errata on this paper is in [21].

[21] Y. Gurevich and J.K. Huggins. *Errata to "The Semantics of the C Programming Language"*, volume 832 of *LNCS*, pages 334 – 336. Springer Verlag, 1994.

[22] N.Wirth H.Moessenboeck. The programming language oberon-2. *Structured Programming*, 12(4):179–195, 1991.

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

[24] D.E. Knuth. Semantics of Context–Free Languages. *Math. Systems Theory*, 2(2):127 – 146, 1968.

[25] P.W. Kutter. Dynamic semantics of the programming language oberon. Technical report, TIK, ETH Züerich, 1996.

[26] P.W. Kutter. Executable Specification of Oberon Using Natural Semantics. Term Work, ETH Zürich, implementation on the Centaur System [12], 1996.

[27] P.W. Kutter. Integration of the Statecharts in Specware and Aspects of Correct Oberon Code Generation. Master's thesis, Kestrel Institute, 1996.

[28] P.W. Kutter and A. Pierantonio. Static Semantics of the Programming Language Oberon. Technical Report July/117, Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, 1996.

[29] P.W. Kutter and A. Pierantonio. Montages: Unified static and dynamic semantics of programming languages. submitted for publication, 1997.

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

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

[32] J. Palsberg and M.I. Schartzbach. *Object-Oriented Type Systems*. Wiley, 1994.

[33] D. Parigot, G. Roussel, E. Duris, and M. Jourdan. Attribute Grammars: a Declarative Functional Language. Technical Report RR 2662, INRIA, Rocquencourt, 1995.

[34] A. Poetzsch-Heffter. Programming Language Specification and Prototyping using the MAX System. In M. Bruynooghe and J. Penjam, editors, *Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 137 – 150. Springer–Verlag, 1993.

[35] A. Poetzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In P. Fritzson, editor, *Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 233 – 247. Springer–Verlag, 1994.

[36] Posegga, J. and Beckert, B. *leanEA: A Poor Man's Evolving Algebra Compiler*, volume 1092 of *LNCS*, pages 64 – 85. Springer Verlag, 1993. A 9-line Plolog interpreter for Evolving Algebras.

[37] A. Pötzsch-Heffter. Developing Efficient Interpreters Based on Formal Language Specifications. In *International Conference on Compiler Construction*, LNCS, pages 233 – 247. Springer–Verlag, 1994.

[38] C. Pusch. . LNCS. Springer Verlag, 1996.

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

[40] G. Schellhorn and W. Ahrendt. Verification of a prolog compiler - first steps with kiv. Technical Report 27, Universität Karlsruhe, Fakultät für Informatik, 1996.

[41] Y.V. Srinivas, R. Jüllig, A. Goldberg, D.R. Smith, L.-M. Gilham, J. McDonald, L. Blaine, C. Green, and R. Waldinger. *Specware Language Manual.* Kestrel Institute, 1996.

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

[43] USA University of Michigan at Ann Arbor. Evolving Algebras Web Page. http://www.eecs.umich.edu/ealgebras/.

[44] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131 – 164. Oxford University Press, 1994.

[45] D. Watt. *Programing Language Concepts and Paradigms.* Prentice Hall, 1990.

[46] N. Wirth. The Programming Language Oberon. *Software – Practice and Experience*, 18(7):671 – 690, 1988.

[47] N. Wirth. personal communication, email from nov.28th, 1996.

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

[49] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.