Specyfikacja i optymalizacja programów w Smalltalku

Marcin Młotkowski

Praca doktorska

Promotor: Prof. Leszek Pacholski

Instytut Informatyki
Uniwersytet Wrocławski
ul. Przesmyckiego 20
51–151 Wrocław
email: marcinm@ii.uni.wroc.pl

Wrocław, 7 lutego 2001

Specification and optimization of the Smalltalk programs

Marcin Młotkowski

Ph.D. Thesis

supervisor: Prof. Leszek Pacholski

Institute of Computer Science
University of Wrocław
ul. Przesmyckiego 20
PL–51–151 Wrocław
email: marcinm@ii.uni.wroc.pl

Wrocław, February 7, 2001

Abstract
Formal semantics of a programming language is an important tool for constructing correct compilers and reasoning about programs and program transformations. In this thesis we study formal semantics of Smalltalk – an object–oriented programming language. We give an operational semantics, which gives a meaning of a program by providing a description of an execution of this program or a virtual machine. Then we show how a program can be transformed to its optimized version, which is provably equivalent to the original program.

Contents

Chapter 1

Introduction

The area of programming languages is one of the most important topics in computer science, since a programming language is a basic tool of each programmer. The quality of this tool directly influences the quality of the final product of programming: a program.

The rudimentary classification of programming languages divides all languages[1] into three basic kinds:

- *imperative languages*, which contain (probably) most programming languages, like Algol, C, Pascal, Cobol etc. We distinguish here an important subfamily of object–oriented languages like C++, Simula, Java or Smalltalk;
- *functional languages* with Lisp and ML;
- *logic programming languages* including the family of Prolog and Prolog–like languages.

We will study object–oriented languages, in particular Smalltalk. We have chosen Smalltalk because of its growing popularity, and since it is the most paradigmatic object–oriented programming language. Most of other object–oriented languages are imperative languages with standard imperative constructions like loops or conditional statements with added object–oriented features. For instance, Simula or C++ are extensions of Algol and C, respectively. In contrast, Smalltalk has been designed as a pure object–oriented language; everything is an object, and the only statements are message expressions, assignments, and return statements. In Smalltalk constructions equivalent to loops and conditional statements can be simulated by methods of special classes. A gentle introduction to Smalltalk will be given in Chapter 2.

Each programming language should have good, i.e. unambiguous definition which describes its syntax and semantics. The definition gives meaning to each language construction allowing writing, reading and understanding programs. The syntax is expressed as a *context–free grammar* which gives the simplest part of the definition. Techniques of generating parsers from the description are well understood and allow almost automatic generation. Semantics is more complicated, because giving meaning to program is a difficult task. A semantics should explain all (or at least many) details important for compiler's implementation. For programmers a verbal

[1] We are interested in higher level languages, i.e. we disregard machine languages, assemblers etc.

description is usually sufficient to learn and use a language, and a verbal description is often used as a definition by compiler constructors. However, such a description usually is not sufficiently precise to express subtle details. A good example of imprecision of natural languages is a definition of Algol 60 which has been written very carefully, but turned out to be ambiguous (see [NBB$^+$60]). Such situations can not be accepted, because it makes it possible to write two different correct (in the sense of a given definition) compilers which generate non–equivalent codes.

In this thesis we study the semantics of languages, the syntax being used as a tool in the definition of semantics. There are three types of semantics of programming languages: operational semantics, denotational semantics, and axiomatic semantics. Each of them represents a different approach to attaching meaning to programs.

The operational semantics in the form given by Hennessy and Plotkin in [HP79] consists of a set of *configurations*, each of which describes a snapshot of computation, and a *transition relation* between configurations, which defines an execution of a given program. The transition relation specifying configuration's change is defined for each language's construct, such as assignment, loop, or conditional statement.

A completely orthogonal approach is taken by the denotational (called sometimes mathematical) semantics proposed by Strachey [Str66]. In this semantics a meaning of a program is given by a *semantics function* which for a given program and perhaps other arguments returns an element of a set called *semantics domain*. This function is defined by induction on the structure of a program. The loop and recursive statements are expressed using *fixed points*.

The axiomatic semantics fixes the meaning of a program by giving its properties expressed in a logic, called *programming logic*. The semantics is defined by proof rules associated to each programming construct. This semantics is designed to provide tools to give a proof of correctness of programs.

In this thesis we study operational semantics of Smalltalk. Our intention is to give a precise description of a computation in Smalltalk. As a language of semantics description we use *Abstract State Machine* (previously known as *evolving algebras*) because of its freedom to choose the level of abstraction. Using this semantics we investigate a formal approach to program transformation and optimization.

1.1 Overview of the thesis

In Chapter 2 we explain basic notions of object–oriented programming using some examples and we give an introduction to Smalltalk. A reader familiar with these topics can skip this chapter.

In Chapter 3 we give a definition of an Abstract State Machine, which we use as a language for defining operational semantics.

Chapter 4 contains a formalization of the structure of a Smalltalk program, and specification of a state of computation of Smalltalk as a set of appropriate universes and functions, called algebra $\mathcal{A}_0$. In Chapter 5 we define a computation of a Smalltalk program as a transition rule $\mathcal{R}_0$ which transforms $\mathcal{A}_0$. Both chapters include results presented in [KMW99].

In Chapter 6 we define transformation of $\mathcal{A}_0$ and $\mathcal{R}_0$ to the so called *continuation* form, which directly expresses control flow. As a result we obtain a simpler but less abstract form of $\mathcal{R}_0$.

In Chapter 7 we consider some techniques to speed-up programs written in object–oriented languages. We give two such techniques: static dispatch and inlining. Both techniques are realized by program transformations. We use semantics to prove equivalence of original and transformed programs. We also discuss limitations of these methods.

Chapter 8 gives an overview of different approaches to defining semantics of programming languages. We also compare other formal definitions of object–oriented programming languages including Smalltalk given by other authors.

In Appendix A we give a context–free grammar defining syntax of methods in Smalltalk which is studied in our thesis.

In Appendix B we give signatures of all algebras and lists of all rules defined and used in our thesis.

Chapter 2

Introduction to Smalltalk

Before giving a formal definition of Smalltalk, we provide an overview of this programming language to allow a reader who is unfamiliar with Smalltalk to understand this thesis. This overview has two parts: the first part is a general introduction to objects; the second explains how Smalltalk realizes the paradigm of object–oriented programming.

2.1 Object–oriented programming

The key concept of object–oriented programming is an object. The object can be considered as a “black–box” which has its own fields (called sometimes *attributes*), and which can execute specified operations called *methods*. From the programming language point of view an object is an element similar to a record, but much more complicated. Instead of giving formal definitions of an object, an attribute and a method, we explain these important notions using an example.

The following representation of a vehicle can serve as an example of an object. It has attributes *'colour'* and *'max–speed'*; and two methods: *'start'* and *'stop'*. There can exist many similar objects, i.e. with the same set of attributes and methods. Informally speaking all such objects “look similar”, belong to the same *class*, which can be regarded as a type of these objects. Using our example we can write, that objects named *'fiat'*, *'opel'* or *'toyota'* belong to a class ***Vehicle***. All these objects have attributes *'colour'* and *'max–speed'*, and the methods *'start'* and *'stop'*. The class ***Vehicle*** is a pattern of object which consists of list of all attributes and methods. If we want to create a new object of this class, e.g. *'chrysler'*, then ***Vehicle*** should be the parameter of this operation. The idea of a class can significantly decrease time of programming: all methods are written only once in the definition of a class and then can be freely used by all objects of the class.

The class ***Vehicle*** represents a general notion of a vehicle with no details concerning e.g. its application (car, truck). We can *specialize* the class ***Vehicle*** creating two new *subclasses*: ***Car*** and ***Truck***. Each of these new classes *inherits* attributes and methods from their *superclass*: ***Vehicle***, but it is possible to add new attributes and methods to each class. For example, we can extend ***Truck*** by the attribute *'capacity'*, and extend ***Car*** by the *'fuel-type'* attribute. We have also possibility to add methods *'load-truck'* and *'unload-truck'* to ***Truck***, and new methods *'open-roof'* and *'close-roof'* to ***Car***. In addition, we can redefine any inherited method in a subclass, e.g. if starting a truck needs some different operations than starting a vehicle, then the method *'start'* can be rewritten in ***Truck***. These extensions affect only the subclass and its objects. Objects of the class ***Truck*** can do *'load-truck'*, but objects of ***Vehicle*** and ***Car*** can not. The relation between a subclass and its superclass constitutes a tree called *class hierarchy*, which can be extended by adding more and more specialized classes according to programmers needs. In many object–oriented languages inheritance from only one class is allowed, but there are languages, e.g. C++, where multiple inheritance is possible.

2.2 Smalltalk

The object–oriented paradigm presented in the previous section can be realized in almost all programming languages using more or less complicated tricks, but full strengths of object–oriented techniques can be used in object–oriented languages.

The structure of Smalltalk is very simple. The language is based on a small number of the following concepts: class, object, inheritance, method and message. This leads to a simple syntax. Roughly speaking, any program is a class hierarchy with class declarations and methods associated to classes.

Expressions

Every method is a sequence of expressions separated by . (dot). Basic expressions are constants and variables, as in other languages, but all constants like 5 or *'string'* are members of appropriate classes, like ***SmallInteger*** or ***String***.

More complex expressions are *message expressions*, which are requests to an object for executing a method[1]. Each message expression consists of an object, called a *receiver*, and a *message*, which consists of a name of a method and optional arguments. Let us consider an example

```
3 + 5 * 2
```

which can be interpreted as following: a message named '+' is sent to the object 3 with argument 5. Sending a message causes an invocation of a method found in a class of a receiver. The method '+' in the class ***SmallInteger*** has its own definition and computes arithmetic interpretation of the expression `'3 + 5'` [2]. The result of computation of this expression is an object 8. In the next step the message '∗' is sent to 8 with argument 2, what gives 16. Message expressions are interpreted from left to right, and since '+' and '∗' are methods, not operators, no precedence is taken under consideration.

The method can have no argument (*unary method*), we simply write it as following

```
5 neg
```

[1]There are also cascaded message expressions, but they are equivalent to a sequence of expressions containing messages to one object.

[2]Of course, this method can be modified like other methods.

what means sending the message *'neg'* to the object 5.

In the case of methods with many arguments, all arguments are separated by so called *keywords*. An example of a method expression using keywords is

```
5 abc: 3 xyz: 6
```

which is interpreted as sending message *'abc:xyz:'* with two arguments 3 and 6 to the object 5.

Methods

Let us give an example of a Smalltalk's method. This method is defined in the class which members are integers. In Smalltalk this class is called ***SmallInteger***.

Example 2.1

```
abc: b  xyz: c
| d |
d ← (b * b - (4 * self * c)) sqrt.
↑(b neg - d)/(2 * self)
```

The first line of the method is a declaration of the name of the method and consists of names of method's parameters: *'abc:xyz:'* is a name of the method [3], *'b'* and *'c'* are parameters of the method. Unlike other languages, in Smalltalk the name of a method is interleaved with names of parameters, and a form of this name determines a number of parameters of the method. E.g. a name *'neg'* means a method with no parameters, and '+', '-' or '<=' denote methods with one argument.

The second line

```
| d |
```

is a declaration of a local variable *'d'*.

The third and fourth lines constitute a body of the method. The body consists of two statements separated by '.' (dot).

We remind here, that arithmetic symbols like '−' or '∗' are methods, not operators. So, *'$b * b$'* is not multiplication in the traditional sense, but it is a message '*' sent to the receiver b, with an argument b. Of course, implementations of all methods named like traditional symbols, such as '+', '∗' or '<=', correspond to the traditional meaning of these symbols. Complex expressions including more than one message, e.g. *'4 * self * c'*, are evaluated from left to right. Therefore in order to obtain desired order of computation, parentheses should be used.

A variable *'self'* has special meaning in Smalltalk. Sometimes if a method is invoked for some receiver, it is necessary to invoke another method for the same receiver in the body of computed method. Therefore in each method an object being a receiver is available as a value of a variable named *'self'*. Notice, that each method can be computed only if it is invoked for some object. Therefore a value of *'self'* is always determined.

The method *'abc:xyz:'* can be used in the following message expression

[3] Names of methods are called *signatures* in Smalltalk.

```
1 abc: 6 xyz: 9
```

where 1 is a receiver of the method, and 6 and 9 are arguments. The receiver 1 belongs to the class ***SmallInteger***. So, a method, which name matches to *'abc:xyz:'* will be searched in this class. Hence a method declared in Example 2.1 will be invoked, and 1, 6 and 9 will be values of *self*, *'a'* and *'b'*, respectively.

In such a case in according to the given principles of evaluation, expressions of the body of the method are computed as follows:

- *'b * b'* is evaluated, i.e. the message *'*'* with argument b is sent to b, what gives 36;
- the expression *'4 * self * c'*, an argument of the message *'–'*, is evaluated:
  - the message *'*'* is sent to 4 with an argument *self*, what gives 4;
  - a result of *'4 * self'* becomes a receiver of the message *'*'* being sent with an argument *'c'*, what gives 36;
- a message *'–'* is sent to the result of *'b * b'* with an argument being a result of *'4 * self * c'*, what gives 0, and 0 is assigned to *'d'*.
- *'b neg'* is evaluated, what gives -6;
- a method *'–'* with argument *'d'* is called for the result of *'b neg'*, what gives -6;
- *'2*self'* is evaluated, what gives 2;
- a message *'/'* is sent to the result of *'b neg - d'* with the result of *'2*self'* as an argument; what gives -3.

A sign *'↑'* in the fourth line means, that a value of the expression following *'↑'* will be returned as a value of the method. So, the value of the message expression *'1 abc: 6 xyz: 9'* is -3.

Blocks

In Smalltalk there is no syntactic construction expressing the conditional or the loop statements known from Algol-like languages. These constructions are simulated using blocks. A block is a special kind of an object, which consists of a list of expressions. Syntactically, block is a Smalltalk expression built of a list of expressions surrounded by square brackets `'['` and `']'`, e.g. `'[2 + 2]'`. For our purposes we call such expressions with square brackets *block expressions*. If a block expression is evaluated, then a new object – block – is created. This block becomes a value of evaluated block expression.

A list of expressions included between `'['` and `']'` is called a *body* of the block expression. E.g. a body of `'[2 + 2. Date today]'` is `'2 + 2. Date today'`. It is worth emphasizing, that evaluation of a block expression creates a new object – block – but the body of this block is left unevaluated.

Block expressions can be used like other expressions, i.e. their values can be assigned to variables or passed as arguments of methods being called. Below we give an example of such a block expression:

```
x ← [ 2 + 2 ].
```

In this example by evaluation of `'[2 + 2]'` a new block, say b, is created. Next, b is assigned to a variable `x`.

A body of a block can be evaluated by special methods. One of them is named *'value'*. Consider an example

```
y ← [ 2 + 2 ] value.
```

In this example first a block, say c, is created. Then the body of c, `'2 + 2'`, is evaluated, what gives 4. Finally, 4 is assigned to a variable `y`.

The possibility of delaying evaluation of expressions by putting them into blocks, allows simulation of loop and conditional statements. The if-statement is simulated as follows: there are two classes in Smalltalk called ***True*** and ***False***. In each of them methods *'ifTrue: ifFalse:'* are defined as follows:

in class ***True***:	in class ***False***:
`ifTrue: trueBl ifFalse: falseBl` `↑trueBl value`	`ifTrue: trueBl ifFalse: falseBl` `↑falseBl value`

where `'↑e'` returns value of expression `e` as a value of a method. Therefore, if expression `'i < 0'` returns an object belonging either to the class ***True*** or to the class ***False***, then we can write an expression

```
(i < 0) ifTrue: [0] ifFalse: [1].
```

which evaluates to 1 if *i* is nonnegative, or to 0 otherwise. Notice, that in this example the body of only one block is evaluated.

Classes

Classes in Smalltalk are also objects. Therefore they can perform methods. Creation of a new object of some class can be done by sending an appropriate message to a class. This method is usually named *'new'*. If ***OrderedCollection*** is a name of a class, then an expression

```
table ← OrderedCollection new.
```

creates a new object of the class ***OrderedCollection***, and assigns this object to a variable `table`.

Now we shall give an example of a class hierarchy. We are going to implement basic notions and operations concerning geometric figures. A declaration[4]:

Object **subclass:** *Figure*
instanceVariableNames: *'xcoord ycoord'*

declares a new class ***Figure*** as a subclass of a class ***Object***[5]. Each object of ***Figure*** has two attributes: *'xcoord'* and *'ycoord'*. Intentionally, objects of ***Figure*** correspond to planar figures, and *'xcoord'* and *'ycoord'* denote a position of a figure on a plane.

The class ***Figure*** does not describe any particular type of geometric figure, but defines properties shared by all planar figures: coordinates. Therefore methods depending on features of particular figures cannot be implemented in ***Figure***, e.g. a method drawing a figure. Although some methods can be defined. Consider for example an algorithm, which moves a figure from one position to another by a given vector (dx, dy). The easiest way to do it, is drawing the figure in the colour of background, changing figure's coordinates, and drawing the figure again but in the colour different from colour of background. This algorithm is implemented by the following method:

```
move: dx move: dy
self draw: BackgroundColour.
xcoord ← xcoord + dx.
ycoord ← ycoord + dy.
self draw: ForegroundColour
```

The variables *'BackgroundColour'* and *'ForegroundColour'* denote global variables storing colours of background and foreground, respectively. Statements

```
self draw: BackgroundColour.
```

and

```
self draw: ForegroundColour
```

draws a figure pointed by *self* in the given colour. Notice, that if a value of *self* is an object of the class ***Figure***, then a method named *'draw:'* cannot be found, because no method named *'draw:'* is defined in ***Figure***. It means, that each call of the method *'draw:'* for an object of the class ***Figure*** raises a runtime error. Recall that the class ***Figure*** represents a general notion of a geometric figure, and objects of this class are not intended to be created.

Now we can define two concrete types of geometric figures: equilateral triangle and rectangle, by the following declarations:

Figure **subclass:** *Triangle*
instanceVariableNames: *'side'*

and

[4] This is slightly modified version of a class declaration in Smalltalk
[5] Each class in Smalltalk is a subclass of the class ***Object***

Figure **subclass:** *Rectangle*
instanceVariableNames: *'a b'*

which introduce two classes ***Triangle*** and ***Rectangle*** as subclasses of ***Figure***. Since objects of the class ***Triangle*** and ***Rectangle*** represent concrete figures, the method *'draw:'* can be defined in each of this class. In the class ***Triangle*** this method has the definition:

```
draw: colour
| pen |
pen ← Pen new.
pen place: xcoord@ycoord.[6]
pen color: colour.
pen north.
pen go: side.
pen turn: 120.
pen go: side.
pen turn: 120.
pen go: side.
pen turn: 120.
```

In the class ***Rectangle*** the method *'draw:'* is implemented as follows:

```
draw: colour
| pen |
pen ← Pen new.
pen place: xcoord@ycoord.
pen color: colour.
pen north.
pen go: a.
pen turn: 90.
pen go: b.
pen turn: 90.
pen go: a.
pen turn: 90.
pen go: b.
pen turn: 90.
```

Pen is a name of library class in Smalltalk. A statement

```
pen ← Pen new.
```

creates an object of the class ***Pen*** and assigns it to a variable *'pen'*. Objects of ***Pen*** can draw on screen in style of turtle graphics. A method *'go:'* moves the pen from its last position to the next leaving a trace of moving. An argument of *'go:'* is a distance between these positions. A direction of moving is determined by a pen direction, and can be changed by methods *'turn:'* or *'north'*.

[6] The expression `xcoord@ycoord` denotes a point.

The expression *'xcoord@ycoord'* denotes an object of a class ***Point***, which represents a planar point of coordinations $(xcoord, ycoord)$.

Since ***Triangle*** and ***Rectangle*** are subclasses of ***Figure***, they inherit all methods and variables from ***Figure***. It means, that the variables *'xcoord'* and *'ycoord'* are available in these classes and can be used in the definition of *'draw:'*.

The method *'move:move:'* is inherited as well, hence can be called for any object of the class ***Triangle***, e.g.

```
t ← Triangle new.
t draw:  ForegroundColour.
t move:  3 move:  -5.
```

In contrast to the objects of the class ***Figure***, this code does not raise a runtime error, because in this case in the fragment of *'move:move:'*

```
self draw:  ForegroundColour.
```

the value of *'self'* is an object being the value of *'t'*, i.e. belonging to the class ***Triangle***, in which the method *'draw:'* is defined.

Later we will present other examples of Smalltalk's expressions and methods.

Chapter 3

Abstract State Machine

In this chapter we give a definition of an Abstract State Machine (ASM), a model of computation for specifying complex systems, as programming languages, algorithms, hardware etc. The definition of ASM was proposed by Y. Gurevich in [Gur91] (ASM is called there *evolving algebras*). Later, in [Gur95] he proposed a revised notion of an ASM admitting some extensions. In [Gur00] he gave a relationship between sequential algorithms and Abstract State Machines. We use only the basic version of the notion, which is sufficient for our purposes.

An Abstract State Machine consists of a universal algebra and a program. The algebra can be regarded as a data structure which describes a state of the considered system. The program is a transition rule of the algebra, which expresses changes of system's state. The notions of an universal algebra and a transition rule are defined in Sections 3.1 and 3.2, respectively. In Section 3.3 we explain the notion of an evolution of an algebra.

3.1 Terms and universal algebras

A *signature* Σ is a set of function symbols (or function names), where each symbol is equipped with a nonnegative integer number called *arity*. Function symbols of arity 0 are called constants. Let

$$\Sigma = \langle \mathsf{Const}_1, \ldots, \mathsf{Const}_n, Fn_1, \ldots, Fn_m \rangle^1$$

be a signature. We assume that $\mathsf{Const}_1, \ldots, \mathsf{Const}_n$ are constants, and $Fn_1, \ldots, Fn_m$ are function symbols of positive arity. Besides symbols from Σ, we use a finite set V of variables.

For given Σ, the set of terms is the smallest set containing V and satisfying the following condition:

if F is a k-ary function symbol in Σ, and $t_1, \ldots, t_k$ are terms, then $F(t_1, \ldots, t_k)$ is a term.

It is clear, that all constants are terms. Terms that do not contain any variables are called *ground terms*.

[1] We use different fonts for constant and function symbols, see Section 3.4.

A universal algebra of a signature Σ is a structure

$$A = \langle U, c_1, \ldots, c_n, f_1, \ldots, f_m \rangle$$

with a nonempty set U, distinguished elements $c_1, \ldots, c_n \in U$ and functions $f_1, \ldots, f_m$. Each of these functions is defined on some Cartesian power of U, and assuming the elements of U as values. There is a correspondence between the signature Σ and the structure A, such that each constant symbol Const_i corresponds to the constant c_i, and each function symbol Fn_i corresponds to the function f_i. Moreover, if Fn_i is a function symbol of arity k, then

$$f_i : U^k \to U$$

i.e. f_i is a function from U^k to U. The set U is called a superuniverse of A.

A superuniverse of ASM is often a union of sets called universes. If it is not explicitly stated, universes are disjoint up to an element undef, whose role is described below. Each universe has a name, and for each universe there exists in Σ a function symbol of arity 1 having the same name as this universe, so, there exists in A a corresponding function with values in the set $\{\mathsf{true}, \mathsf{false}\}$. We shall also use names of universes as names of sets. Hence, we write $t \in \textsc{Universe}$ instead of $\textsc{Universe}(t) = \mathsf{true}$.

All functions in ASM are total, but sometimes it is useful to consider partial functions. In order to do it we introduce a distinguished value undef representing undefined value, i.e. we replace each partial function

$$f : X \to U,\ X \subseteq U^k$$

by a total function by putting

$$f(\overline{x}) = \mathsf{undef}$$

for $\overline{x} \notin X$. We assume that all functions are strict, i.e. if $\overline{x} = x_1, \ldots, x_n$ and one of x_i is undef, then $f(x_1, \ldots, x_n) = \mathsf{undef}$.

As we have said earlier, each constant symbol in the signature Σ denotes a constant in A, i.e. an element of the superuniverse U, and each function symbol has its own corresponding function in A. We define a value of each ground term t by induction:

- if t is a constant symbol Const_k, then value of t is c_k;
- if t is a term of the form $Fn(t_1, \ldots, t_k)$, the function symbol Fn corresponds to a function f, values of the terms $t_1, \ldots, t_k$ are elements $a_1, \ldots, a_k \in U$, respectively, and $f(a_1, \ldots, a_k) = a$, then value of t is a.

This definition can be easily extended to the form which defines values of terms including variables if valuation of variables is given.

In ASM terms with variables are used only in special context. In Section 3.3 we shall explain how we use and interpret variables.

3.2 Transition rules

The universal algebra defined in the previous section can describe only "static world", but we need more to express changes of complex systems. Therefore we allow modifications of the algebra, i.e. modifications of functions: for a given function f and its arguments the value of f can be changed. In this way we can change values of terms. It can be done by an operation called *update rule* which has the form:

$$t_1 := t_2$$

The informal meaning of such update rule is the following: first t_2 is evaluated, then this value is assigned to t_1.

The second type of operation available in ASM is an *extend rule*. It allows adding new elements to universes if necessary. The extend rule has a form

extend UNIVERSE **with** *var*
    update rules
endextend

By this operation a new element is added to a universe UNIVERSE. During the execution of this rule, this new element is a value of a variable *var* which by update rules can be assigned as value of some terms in the body of extend rule.

Executions of some update rules make sense only if some conditions are satisfied. We introduce a *conditional rule* of the form

if cond_1 **then**
    R_1
elseif cond_2 **then**
    R_2
elseif cond_3 **then**
    R_3
...
else
    R_n
endif

where R_i are sequences of updates, extend rules and other conditional rules. The conditions $cond_i$ are (in)equalities of terms interpreted as (in)equalities of values of these terms.

A *transition rule* is a sequence of updates, extend rules and conditional rules. Order of rules in this sequence is not important. Formally the syntax of transition rule is described by the following grammar $\mathcal{G}$:

$$\begin{array}{lcl} \langle rule \rangle & := & \langle rule \rangle \; \langle rule \rangle \\ & | & \langle update\_rule \rangle \\ & | & \langle conditional\_rule \rangle \end{array}$$

$$
\begin{array}{lcl}
 & | & \langle \mathit{extend\_rule} \rangle \\
 & | & \langle \mathit{name\_of\_rule} \rangle \\
 & | & \langle \mathit{empty\_rule} \rangle \\
\\
\langle \mathit{update\_rule} \rangle & := & \langle \mathit{function\_name} \rangle \text{ '(' } \langle \mathit{term} \rangle \textbf{ [} \text{','} \langle \mathit{term} \rangle \textbf{]}^{*} \text{ ')' ':=' } \langle \mathit{term} \rangle \\
 & | & \langle \mathit{constant} \rangle \text{ ':=' } \langle \mathit{term} \rangle \\
 & | & \langle \mathit{name\_of\_term} \rangle \text{ ':=' } \langle \mathit{term} \rangle \\
\\
\langle \mathit{conditional\_rule} \rangle & := & \text{'}\textbf{if}\text{' } \textbf{[} \langle \mathit{condition} \rangle \text{ '}\textbf{then}\text{' } \langle \mathit{rule} \rangle \text{ '}\textbf{elseif}\text{' } \textbf{]}^{*} \langle \mathit{condition} \rangle \text{ '}\textbf{then}\text{' } \langle \mathit{rule} \rangle \\
 & & \textbf{[} \text{'}\textbf{else}\text{' } \langle \mathit{rule} \rangle \textbf{] } \text{'}\textbf{endif}\text{'} \\
\\
\langle \mathit{extend\_rule} \rangle & := & \text{'}\textbf{extend}\text{' } \langle \mathit{universe} \rangle \text{ '}\textbf{with}\text{' } \langle \mathit{variable} \rangle \langle \mathit{rule} \rangle \text{ '}\textbf{endextend}\text{'} \\
\\
\langle \mathit{empty\_rule} \rangle & := & \\
\\
\langle \mathit{condition} \rangle & := & \langle \mathit{term} \rangle \text{ '=' } \langle \mathit{term} \rangle \\
 & | & \langle \mathit{term} \rangle \text{ '}\neq\text{' } \langle \mathit{term} \rangle \\
 & | & \langle \mathit{name\_of\_condition} \rangle \\
\\
\langle \mathit{term} \rangle & := & \langle \mathit{variable} \rangle \\
 & | & \langle \mathit{constant} \rangle \\
 & | & \langle \mathit{name\_of\_term} \rangle \\
 & | & \langle \mathit{function\_name} \rangle \text{ '(' } \langle \mathit{term} \rangle \textbf{ [} \text{','} \langle \mathit{term} \rangle \textbf{]}^{*} \text{ ')'}
\end{array}
$$

Words with angle brackets are nonterminal symbols. The other symbols are terminal. Additionally we assume that for a given signature Σ terms derived in the grammar $\mathcal{G}$ are consistent with Σ, i.e. each k-ary function symbol has k arguments.

In the case of complex terms, conditions or rules, it is convenient to name such constructions and use their names in place of their full forms. Therefore in $\mathcal{G}$ we allow to use names of terms, conditions and rules. In particular, each transition rule can be written as one rule, but in case of complex ASM, such a large rule is difficult to read, analyze and understand. Names of rules will be defined as follows:

Rule NAME_OF_RULE
 R
EndRule

where NAME_OF_RULE is a name of a rule R. The form of defining names of terms and conditions will be given later.

Not all rules defined by the grammar $\mathcal{G}$ make sense. The problem is caused by terms with variables, since variables have no fixed value independent of the context, hence it is not possible to compute values of such terms. Therefore we define and use only *programs*:

Definition 3.1 *A* program *is a transition rule which satisfy the following condition: each variable* var *occurs only between the preceding words* **extend** UNIVERSE **with** *var and the corresponding word* **endextend** *for some* UNIVERSE.

3.3 Transition of an algebra

In ASM all transitions of algebras are realized by execution of programs. The description of rules given in the previous section is only informal, therefore in this section we give a formal definition of transition of an algebra.

Let A be an algebra over a signature Σ and R be a program. The pair $\langle A, R\rangle$ generates a new algebra B over signature Σ. The algebra B is called the result of transition of A according to the program R.

First we define the meaning of conditions: a condition $t_1 = t_2$ is true in A, if values of t_1 and t_2 are equal in A. Otherwise, this condition is false. Conversely, the condition $t_1 \neq t_2$ is true in A if and only if terms t_1 and t_2 denote different elements in A. If a condition has a form of a name of a condition, then it is true exactly when the condition given in the definition of this name is true.

The interpretation of conditions in conditional rules in A determines a set $E_A(R)$ of applicable update rules, which is a subset of update rules of R. A transition of the algebra A depends on $E_A(R)$.

Definition 3.2 *For a given algebra A and a transition rule R, a set set $E_A(R)$ is defined in according to the following formulas:*

(1) if R is an empty rule, then $E_A(R) = \emptyset$,

(2) if $R = R_1R_2$, then $E_A(R) = E_A(R_1) \cup E_A(R_2)$,

(3) if R is a name of a rule R_1, then $E_A(R) = E_A(R_1)$,

(4) if R is an update rule, then $E_A(R) = \{R\}$,

(5) if R = **if** <u>*cond*</u> **then** R_1 **else** R_2 **endif** *and the condition* <u>*cond*</u> *is true in A, then $E_A(R) = E_A(R_1)$; otherwise $E_A(R) = E_A(R_2)$; a rule* **if** <u>*cond*</u> **then** R_1 **endif** *is equivalent to* **if** <u>*cond*</u> **then** R_1 **else endif***;*

(6) if R = **if** <u>*cond*</u>$_1$ **then** R_1 **elseif** <u>*cond*</u>$_2$ **then** R_2 … **endif** *and the condition* <u>*cond*</u>$_1$ *is true in A, then $E_A(R) = E_A(R_1)$; otherwise $E_A(R) = E_A($***if** <u>*cond*</u>$_2$ **then** R_2 … **endif**$)$*,*

(7) if R = **extend** UNIVERSE **with** var R_1 **endextend***, then*

$$E_A(R) = E_A(R_1) \cup \{\text{RESERVE}(\mathsf{var}) := \mathsf{false}, \text{UNIVERSE}(\mathsf{var}) := \mathsf{true}\}.$$

If R is a program, then each variable `var` may occur only in a rule R_1 included in

extend UNIVERSE **with** var
    R_1
endextend

Otherwise, `var` has no interpretation.

A transition of an Abstract State Machine according to a rule **extend** should, first of all, cause an extension of the universe UNIVERSE by a new element. In the definition given in [Gur95] this element comes from the universe RESERVE disjoint with all other universes. Hence, we interpret var as a choice of an element r of RESERVE. If R_1 is evaulated, then `var` has the value equal to r. Additionally, r is discarded from RESERVE and added to UNIVERSE. After the element r has been added to UNIVERSES, all rules in R_1 should be executed.

The set of update rules $E_A(R)$ can be applied to the algebra A changing functions of A. Each update rule changes A in the following way. Let

$$Fn\,(t_1,\ldots,t_k) := t$$

be an update rule, let $a_1,\ldots,a_k,a \in U$ be, respectively, values of terms $t_1,\ldots,t_k,t$ in the algebra A. We interpret this rule as a command of changing A by changing the function with name Fn. During the transition of A a function f corresponding to Fn is replaced by the function f' defined by the formula

$$f'(x_1,\ldots,x_k) = \begin{cases} a & \text{if } \langle x_1,\ldots,x_k\rangle = \langle a_1,\ldots,a_k\rangle, \\ f(x_1,\ldots,x_k) & \text{otherwise.} \end{cases}$$

We define in a similar way the transition of A according to the rule $\mathsf{Const} := t$. In this case an interpretation of Const is changed. In the transformed algebra, this constant denotes the element which is the value of the term t in A.

If we intend to transform an algebra A according to a program R, then first we find the set $E_A(R)$ of update rules of R which can be executed in A. Then for each rule

$$Fn\,(t_1,\ldots,t_k) := t \in E_A(R)$$

we compute values of terms $t_1,\ldots,t_k,t$ in the algebra A (for rule $\mathsf{Const} := t$ we compute the value of the term t). Next, we perform the transition described in the previous paragraph for all rules from $E_A(R)$. An algebra B obtained in this way is a result of transition of the algebra A according to the rule R.

It is easy to see that the definition of B may depend on order, in which rules from the set $E_A(R)$ are executed. It happens, when in $E_A(R)$ there are two rules in different ways changing values of the same function for the same argument. Therefore we assume that programs should be written in such a way that this situation does not happen. The example given below shows how correctness of the definition of the algebra B can be ensured. Let R be the following rule:

if $\underline{cond}_1$ **then**
  if $\underline{cond}_{1,1}$ **then**
    $\mathrm{R}_{1,1}$
  else
    $\mathrm{R}_{1,2}$
  endif
elseif $\underline{cond}_2$ **then**
  if $\underline{cond}_{2,1}$ **then**
    $\mathrm{R}_{2,1}$
  endif
endif

R is a complicated conditional rule. It is clear that $E_A(R)$ is always one of the sets $E_A(R_{i,j})$, so it suffices to ensure that all rules $R_{i,j}$ describe B uniquely.

In Chapter 5 we define a complex rule $\mathcal{R}_0$, which describes the computation of a Smalltalk program. The rule $\mathcal{R}_0$ is constructed in such a way, that given an algebra $\mathcal{A}$, $\mathcal{R}_0$ uniquely determines the result of applying $\mathcal{R}_0$ to $\mathcal{A}$, which represents one step of computation of Smalltalk of the program and data represented by $\mathcal{A}$.

Having an algebra $\mathcal{A}$ and a rule $\mathcal{R}$, we can construct a sequence

$$\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \ldots$$

of algebras such that $\mathcal{A}_1 = \mathcal{A}$, and for every positive integer i, the algebra $\mathcal{A}_{i+1}$ is the result of transition of the algebra $\mathcal{A}_i$ according to the rule $\mathcal{R}$. If there exists i such that $E_{\mathcal{R}}(\mathcal{A}_i) = \emptyset$, then $\mathcal{A}_i = \mathcal{A}_j$ for $i \leq j$. We call $\mathcal{A}_i$ the result of evolution of $\mathcal{A}$ according to $\mathcal{R}$.

3.4 Notational conventions

When defining the Abstract State Machine $\langle \mathcal{A}_0, \mathcal{R}_0 \rangle$, we write some kinds of symbols in a characteristic way with or without the use of capital letters and using fixed fonts. We assume that names of universes are written as the word UNIVERSE, function names as *Function*, names of static constants (having fixed interpretation) as constant, names of the other constants as Constant, names of variables as variable, names of rules as RULE, names of terms as *Term* and names of conditions as *condition*.

When we introduce names of universes, constants and functions of an algebra we sometimes give some constraints on the values that functions and constants can assume during any evaluation of $\mathcal{A}_0$. When introducing names of universes we give constraints of the form

$$\mathrm{UNIVERSE}_1 \subseteq \mathrm{UNIVERSE}_2.$$

In the case of constants, corresponding constraints have the form

$$\mathsf{Const} \in \mathrm{UNIVERSE}.$$

Similar constraints referring to functions have the form

$$Fn : \textsc{Universe}_1 \times \textsc{Universe}_2 \to \textsc{Universe}.$$

The meaning of two first constraint is clear. The third constraint is true in some algebra, if the function corresponding to Fn has value undef outside

$$\textsc{Universe}_1 \times \textsc{Universe}_2,$$

and it maps its domain into the set $\textsc{Universe}$. When giving these constraint above we require that they are true in $\mathcal{A}_0$ and are preserved during the transition according to $\mathcal{R}_0$.

Chapter 4

Representation of Smalltalk in ASM

4.1 Introduction

In this chapter we give a formal operational semantics of the programming language Smalltalk. First we define an abstract representation of programs of Smalltalk: each notion such as object, method or expression is represented by an element of an appropriate universe, and relationships among these notions are represented by functions defined on universes. In the second part of this chapter we define an abstract representation of a state of computation.

As we have mentioned in Chapter 2, Smalltalk has a very simple structure: each Smalltalk program is a class hierarchy with sets of methods assigned to classes. The syntax of body of methods is also very simple: there are only assignments, return statements, and message expressions. Loop or conditional statements are simulated by objects of some classes. On the other hand, definition of Smalltalk given by its inventors in [GR89] contains not only definition of the language, but also description of a large library including classes implementing various input/output operations, and other useful classes.

Therefore we have to decide which part of Smalltalk will be covered by our definition. Our primary aim is to describe a computation in Smalltalk: evaluation of expression, dynamic dispatch of methods, method invocation, and termination. Therefore our semantics describes class hierarchy, inheritance of methods, variables, and methods. We do not consider library classes. Since all arithmetic or input/output operations are defined in these classes, we do not describe them. We make here one exception. Although blocks are objects of some class, they play important role in computation: all loop and conditional statements are simulated by means of blocks. Without blocks we are able to implement only trivial computations. We do not consider here details concerning memory management: new object creation and garbage collection.

We consider Smalltalk-80 described in [GR89] as a standard of Smalltalk, but we also refer to the proposed ANSI standard of Smalltalk ([ANS97]).

4.1.1 Formal model

As a formal model of operational semantics we use Abstract State Machine presented in Chapter 3. We define a pair $\langle \mathcal{A}_0, \mathcal{R}_0 \rangle$ expressing semantics of Smalltalk. The definition of $\langle \mathcal{A}_0, \mathcal{R}_0 \rangle$

has two parts. First we describe universes and functions which constitute the universal algebra $\mathcal{A}_0$. Section 4.2 shows how objects, classes and methods are represented, and also presents relationships between them. Section 4.3 gives representation of Smalltalk expressions in $\mathcal{A}_0$.

The second part of the definition is given in sections 4.5 and 5 and consists of a description of a virtual machine: universes and functions expressing states of this machine together with the transition rules $\mathcal{R}_0$ modifying these states.

In Appendix A we give the syntax of Smalltalk which is used in the definition of $\mathcal{A}_0$.

4.2 Basic universes and functions

In this section we define some universes and functions of the algebra $\mathcal{A}_0$ which model fundamental concepts of Smalltalk: objects, classes, variables, methods, and relations among them. Other important universes and functions concerning expressions and states are defined in separated sections because of their complex definition.

4.2.1 Names and lists of names

First we define universes NAMES and LISTS_OF_NAMES and related functions and constants. These universes and functions do not change during the transition of the algebra. Other universes and functions may be changed and their evolution is described by appropriate rules.

The set NAMES contains all names admitted in Smalltalk, i.e. signatures of methods and names of arguments, variables, pseudovariables, constants, etc. Sometimes, if it is necessary, we will distinguish elements of this universe writing them in quotes, e.g. *'abc'* $\in$ NAMES.

All finite sequences of elements of NAMES belong to the set LISTS_OF_NAMES. If $word_1$, $word_2$, $\ldots$, $word_n$ are elements of NAMES, then $[word_1\ word_2\ \ldots\ word_n]$ denotes a sequence of $word_1$, $word_2$, $\ldots$, $word_n$, which is a member of LISTS_OF_NAMES. Elements of this set are called lists. The symbol $[\,]$ denotes the empty list.

We assume that in $\mathcal{A}_0$ there are two functions defined on the set LISTS_OF_NAMES:

First : LISTS_OF_NAMES $\rightarrow$ NAMES,

But_first : LISTS_OF_NAMES $\rightarrow$ LISTS_OF_NAMES

and a constant

empty $\in$ LISTS_OF_NAMES.

For a given list L, the function *First* returns the first element of L. Hence

$$First\,([word_1\ word_2\ \ldots\ word_n]) = word_1.$$

But_first (L) is a sequence made by removing the first word from L. So

$$But\_first\,([word_1\ word_2\ \ldots\ word_n]) = [word_2\ \ldots\ word_n].$$

We assume also that values of these functions on the empty list are undefined. The constant empty denotes always the empty list.

We shall also use a relation with the name EQUAL_LENGTHS. This name in the algebra $\mathcal{A}_0$ will denote the set of all pairs of elements of LISTS_OF_NAMES of equal lengths. Hence,

$$\text{EQUAL\_LENGTHS} = \{(x,y) \in \text{LISTS\_OF\_NAMES}^2 : \text{the lengths of } x \text{ and } y \text{ are equal}\}.$$

4.2.2 Classes and objects

Each Smalltalk object has its own variables called either *object variables* or *instance variables*. Each object can perform operations called *methods*. Different objects have different object variables. The number of variables used by an object, names of these variables, and also methods performed by an object are determined by the class of this object.

Some objects are similar to each other, have common methods, and have instance variables with the same variable names. If objects are similar, then we can say that they belong to the same class. In Smalltalk classes are also objects, and each new object is created by appropriate class using informations stored in the class. The notion of the class is a relation between some object O and other object C, being the class of O. The class C knows methods performed by objects of C, and names of fields of these objects. Classes have also special methods allowing to create new objects (differences between methods performed by objects of some class and creating objects of a given class are not essential in this chapter). Hence, the notion of a class of an object is a relation between an object O and another object C being a class. This relation is unique, i.e. for each object its class is uniquely defined. An object of a class C also is called an instance of C.

A Smalltalk programmer has in his disposal a set of classes containing basic classes created by authors of the system, modified and completed during an earlier work of the programmer. Usually in the system there are objects which are not classes. All classes and objects are represented in $\mathcal{A}_0$ by elements of the universe OBJECTS and its subuniverse CLASSES (thus CLASSES $\subseteq$ OBJECTS). It is not important here how these elements are represented. All essential features of both objects and classes will be consequences of properties of functions defined on these universes. We may imagine that OBJECTS and CLASSES have some elements representing objects and classes respectively. We identify these elements with objects and classes.

The relationship between an object and its class is expressed in $\mathcal{A}_0$ by a function

$$\mathit{Class} : \text{OBJECTS} \to \text{CLASSES}.$$

Class is defined for all arguments from the set OBJECTS. For each $O \in$ OBJECTS and $C \in$ CLASSES the equality $\mathit{Class}(O) = C$ holds if and only if C represents the class of the object represented by O.

The class of an object determines almost all properties of an object. Usually if we know the class of an object, then we can get full information about methods performed by this object and names of its fields. It is not possible in $\mathcal{A}_0$. But it is not needed, because our Abstract State Machine is used only to explain how Smalltalk methods are performed.

In the Smalltalk system there exists an object called *nil*. Informally, if no value has been assigned to a variable, then its value is *nil* by default. We introduce a constant

$$\mathsf{nil} \in \textsc{Objects}$$

which represents this object.

4.2.3 Variables

The notion of a variable in Smalltalk is not substantially different from one known from other programming languages. As usual a variable has a value and is identified by a name. All variables in Smalltalk are of the same type in the sense that every object may be a value of any variable but, of course, particular objects belong to different classes and have different sizes.

For our purposes we emphasize here the difference between two notions: variable and variable name. In order to make it more clear, we give two examples of methods declared for objects in the class **SmallInteger**.

Example 4.1

```
arg: a arg: b arg: c
| i |
i ← (b*b) - (4*a*c).
↑i
```

Example 4.2

```
fact
| i |
(self > 1)
   ifTrue: [i ← (self - 1) fact * self]
   ifFalse: [i ← 1].
↑i
```

The method in Example 4.1 has three arguments: *a*, *b* and *c*, and computes expression `'b*b - (4*a*c)'`, which is assigned to *i*, and then returned. The method in Example 4.2 has no argument (`fact` is a name or *signature* of the method), and computes factorial of some number. *self* denotes a *receiver* of the method: the object which receives the method `fact`.

In both methods two local variables *i* are declared and used, but the variables are not the same, of course. Therefore name of variable does not denote variable and its value uniquely.

Let us consider the second method in more details. To compute factorial, the method `fact` is called recursively several times, and each call uses the variable *i*. Again, *i* does not denote here only one variable, but a family of variables, each named *i*.

Summing up, our intention is to show that during computation several variables with the same name can exist simultaneously. Therefore, in our algebra $\mathcal{A}_0$ we have to distinguish variables from their names. In order to do it we define an universe whose elements correspond to variables, and then we explain, how names of variables correspond to variables.

We assume that for any variable there exists a corresponding element of the universe $\textsc{Variables}$. A function

$$\mathit{Variable\_value} : \textsc{Variables} \rightarrow \textsc{Objects}$$

expresses relationship between a variable and its value. We assume additionally that *Variable_value* is defined for all arguments belonging to VARIABLES. Now, we shall explain how to find an element of VARIABLES corresponding to a given name.

4.2.3.1 Types of variables

There are six kinds of variables in Smalltalk:

- shared variables which divide into
  - global variables,
  - class variables,
  - pool variables,
- private variables which split up into
  - object variables,
  - local variables which divide into
    * temporary variables, including variables corresponding to arguments,
    * pseudovariables.

In Smalltalk–80, there is the obligatory principle which allows to distinguish names of shared variables from the others. According to this principle, a name of shared variable starts with a capital letter[1]. To use this principle, we insert in $\mathcal{A}_0$ a subuniverse SHARED_VARIABLE_NAMES $\subseteq$ NAMES containing all words which may serve as names of shared variables. Names of private variables belong to NAMES $\setminus$ SHARED_VARIABLE_NAMES.

To express relationship between names of variables and elements of the universe VARIABLES, we introduce a function for each kind of variables:

$$\mathit{Global\_variable} : \textsc{Shared\_variable\_names} \rightarrow \textsc{Variables}$$

$$\mathit{Class\_variable} : \textsc{Shared\_variable\_names} \times \textsc{Classes} \rightarrow \textsc{Variables}$$

$$\mathit{Pool\_variable} : \textsc{Shared\_variable\_names} \times \textsc{Names} \rightarrow \textsc{Variables}$$

$$\mathit{Object\_variable} : \textsc{Names} \times \textsc{Objects} \rightarrow \textsc{Variables}$$

$$\mathit{Temporary\_variable} : \textsc{Names} \times \textsc{States} \rightarrow \textsc{Variables}$$

$$\mathit{Pseudovariable} : \textsc{Names} \rightarrow \textsc{Variables}$$

where a universe STATES represents a state of the computation in our algebra $\mathcal{A}_0$. Detailed information about this universe are given in Section 4.5.2.

These functions together with *Variable_value* describe relationship between a variable name of an appropriate kind and a value of this variable. Any variable, except of global variables, is determined by a name and some additional parameter.

[1] This principle is not mentioned in [ANS97].

4.2.3.2 Shared variables

Global variables in Smalltalk are available to all objects and any object can change a value of any global variable. There exists at most one global variable with a given name, therefore to determine global variable one needs only its name. Hence the function *Global_variable* has only one argument: a name of a global variable.

Objects of the same class have access to common variables called class variables. Each of these variables is defined in the objects' class. In every class, there exists at most one variable with a given name, but – up to limitations described in Section 4.2.5 – in different classes there can exist different variables with the same name. In the algebra $\mathcal{A}_0$ class variables are values of the function *Class_variable*, where the first argument is a name of a variable, the second one is an element of CLASSES representing the class in which this variable is defined.

All other shared variables are called *pool variables* and are collected in *dictionaries*, which are sets of variables. Dictionaries are identified by their names. It is assumed that for a given name there exists at most one dictionary, and in every dictionary there is at most one variable identified by a given name.

In Smalltalk, each object (i.e. its methods) can use pool variables from a given dictionary if and only if usage of the dictionary is declared in the object's class.

It is possible that some object uses two dictionaries and in each of them a variable with the same name W is declared. In this case an interpretation of W depends on the order of names of dictionaries declared in teh class of the object. It is defined by the rule SHARED_VARIABLE in Section 5.4.2.

A list of dictionary names assigned to a class represented by $C \in$ CLASSES and used by objects of this class is a value of a function

$$\mathit{Shared\_pools} : \text{CLASSES} \rightarrow \text{LISTS\_OF\_NAMES}$$

computed for the argument C. The function *Shared_pools* is defined for all arguments from CLASSES. If a class has no dictionary, then the empty list is the value of *Shared_pools* assigned to it.

In order to indicate an element of the universe VARIABLES which represents a pool variable, the function *Pool_variable* as the first argument takes a name of variable, and the second argument is a name of dictionary.

4.2.3.3 Private variables

Every object has a fragment of the memory available only for this object and partitioned into object variables. Both the number and names of object variables are determined by the class of the object. In $\mathcal{A}_0$ we have access to elements of VARIABLES denoting object variables by the function *Object_variable*. If an object variable of the name W exists for some object, then *Object_variable* assigns an element of VARIABLES representing this object variable to both the name W and the element of OBJECTS representing this object. This function should satisfy some additional conditions. First of all, for each word $W \in$ NAMES and for each $O_1, O_2 \in$ OBJECTS such that

$$\mathit{Class}\,(O_1) = \mathit{Class}\,(O_2)$$

Object_variable (W, O_1) is defined if and only if *Object_variable* (W, O_2) is defined. This condition corresponds to the requirement that objects of the same class have variables with the same name. The remaining conditions concerning *Object_variable* are given in Section 4.2.5 on inheritance.

As we have mentioned earlier, all names of shared variables begin with capital letter, and names of all private variables begin with lower case.

Other private variables are created as temporary variables, which are local variables of method and arguments of blocks and methods. They are created after invoking a method and exist during method execution. If the method is performed many times, e.g. recursively, then each time a new variable is created with the same name. Each of these variables is identified by a name and a *state* of computation.

In order to explain usage of temporary variables, we have in $\mathcal{A}_0$ two functions. Values of the function *Temporary_variable* represents temporary variables, and both block and method arguments. If *Temporary_variable* is defined for a pair $(W,S) \in$ NAMES $\times$ STATES, then W is the name of a temporary variable declared in the method performed in the state S or the name of an argument of this method, or the argument name of a block defined by this method. *Temporary_variable* (W,S) represents a variable of the name W created in the state S.

In the given classification, pseudovariables are local variables too, although some of them always exist. Pseudovariables cannot appear on the left hand side of the assignment. Method parameters are classified as pseudovariables. Elements of VARIABLES corresponding to them are values of *Temporary_variable*. Usually there exist also pseudovariables named *nil*, *true*, *false*, *self* and *super*.

Variables *nil*, *true* and *false* are like constants. Their values are fixed object and are not changed. These variables exist since the begin of each computation (so it is not necessary to declare them) and are available for all objects. In the algebra $\mathcal{A}_0$ we have the function *Pseudovariable* allowing to express relationship between names and pseudovariables of the second kind. It is defined for words W different from *'self'* and *'super'* and recognized as names of pseudovariables. If *Pseudovariable* (W) is defined, then *Variable_value* (*Pseudovariable* (W)) is equal to the suitable object representing the value of the pseudovariable W. For instance, *Variable_value* (*Pseudovariable* $('nil')$) $=$ nil.

Pseudovariables *self* and *super* are used similarly. They also do not need to be declared. These variables play a special role in Smalltalk. Their values are evaluated in a fixed way during execution of the method, and the use of *super* has some additional consequences. More details concerning *self* and *super* are given in Section 5.4.3.

We introduce two constant symbols self and super, which values are *'self'* and *'super'* belonging to the universe NAMES.

4.2.4 Methods

Programmers building applications in Smalltalk may define classes and methods. The methods are defined in a source form and may refer to other methods. Even most standard methods of Smalltalk-80 and some part of Smalltalk itself are written in Smalltalk. Methods given in a source form may be named *ordinary*. Of course, there exist methods which do not use other methods, and which are the base of the system. Such methods are not expressed in Smalltalk. They are called *primitive*. Usually, these methods make access both to hardware (keyboard, display, etc.) and to the inside of the implementation of Smalltalk possible. We distinguish some kind of primitive methods called *block methods*. Block methods play an important role and are not expressed in Smalltalk. We define them explicitly in $\mathcal{R}_0$. Both ordinary and block methods are represented in the algebra $\mathcal{A}_0$ as elements of a universe METHODS.

A text of a Smalltalk method consists of three parts: a header with a signature and argument names, a declaration of temporary variables, and a body of a method. A signature of a method is used to identify a method and it is a word, so it belongs to NAMES. Each Smalltalk method is defined as a method of some class. The class of a method is determined by an object, which executes the method. It is usually the object of the class of the method, but the method can be also executed by objects of other classes. Objects that can execute a method are defined in Section 4.2.5.

In this section and Section 4.3 we give a complex definition. To make it more clear, every fragment of the definition is illustrated by examples which will be sometimes associated with the following method:

Example 4.3
```
a: a  b: b  c: c
| d |
d ← ((b*b) - (4*a*c)) sqrt.
↑(b neg - d)/(2*a)
```

The word *'a:b:c:'* is its signature. This method has three arguments named *a*, *b* and *c* and one temporary variable *d* declared in its definition. The word `'d ← ((b*b) - (4*a*c)) sqrt.↑(b neg - d)/(2*a)'` is the body of this method.

To represent methods in $\mathcal{A}_0$, we first define a function

$$\mathit{Method} : \text{NAMES} \times \text{CLASSES} \to \text{METHODS}.$$

This function is defined for $W \in$ NAMES and $C \in$ CLASSES, if there exists a method with the signature W in the class represented by C. If *Method* is defined for (W,C), then its value is an element representing the method of the signature W in the class C. This element could be the method or another element which we may identify with this method. This function has different values for any different arguments, because in any Smalltalk class there exists at most one method of a given signature.

We define four functions *Method_class*, *Argument_names*, *Temporary_names* and *First_statement* on the universe METHODS. It is easy to define first two of them. The first function

$$\mathit{Method\_class} : \text{METHODS} \to \text{CLASSES}$$

for any argument (W,C) satisfies the following equation:

$$\mathit{Method\_class}\,(\mathit{Method}\,(W,C)) = C.$$

So, it describes the relationship between a method and its class. The second one

$$Argument\_names : \text{METHODS} \to \text{LISTS\_OF\_NAMES},$$

assigns a list of names of arguments of a method M to M (or to the element representing M). In the list $Argument\_names\,(M)$ names are given in the same order as in the text of M. We assume that $Argument\_names$ denotes a total function. It is easy to notice that a name of a Smalltalk method determines the number of its arguments. It implies that functions $Method$ and $Argument\_names$ should satisfy the following condition: if $Method\,(W,C)$ is defined, then the length of $Argument\_names\,(Method\,(W,C))$ is the number of arguments of method called W. This condition is obvious for ordinary methods. We assume it also for the block method $Method\,(W,C)$. In this case, names of method arguments do not play any role, but in this way we remember the number of method arguments.

The function

$$Temporary\_names : \text{METHODS} \to \text{LISTS\_OF\_NAMES}$$

assigns a list of names of both local variables declared in the method M and arguments of blocks occurring in M to an element representing the method M. We will discuss its definition in Section 4.3.4.

The function *First_statement*

$$First\_statement : \text{METHODS} \to \text{EXPRESSIONS}$$

assigns an element $e \in \text{EXPRESSIONS}$, which represent first expression of a method M, to M. Details are given in Section 4.3.4.

Let us notice that if $M \in \text{METHODS}$ represents the method of Example 4.3, then

$$Argument\_names\,(M) = [a\ b\ c]$$

and

$$Temporary\_names\,(M) = [d].$$

The second equality follows from the fact that the method M does not contain any definition of block.

Details concerning block methods are given in Section 4.5.3.

4.2.5 Inheritance of methods and variables

A Smalltalk programmer can define new classes. Each class defined by a programmer is specified as a subclass of an existing class. If a class C_1 is a subclass of a class C_2, then C_2 is called a superclass of C_1. We assume that every class defined in Smalltalk has its superclass, except the class which is a superclass of all other classes. Classes in Smalltalk together with the notion of a superclass form a hierarchy, which can be represented as a set of trees, and so there may be more classes without superclasses. We assume that the hierarchy of classes consists of one tree. However, in real systems there may be more trees. The number of these trees plays no role in later considerations.

To describe the hierarchy of classes, we introduce a function

$$Superclass : \text{CLASSES} \to \text{CLASSES}.$$

This function is defined in such a way that for any elements $C_1, C_2 \in$ CLASSES the equality $Superclass\,(C_1) = C_2$ holds if C_2 represents the superclass of the class represented by C_1 (or equivalently: if C_1 represents a subclass of the class represented by C_2). If a class represented by C has no superclass, then we put

$$Superclass\,(C) = \mathsf{undef}$$

We have defined two different ways of creating new objects using a class C. The first one, defined in Section 4.2.2, allows creation an object O using C as a pattern. Then C is the class of O. The second method allows to create a new object C', which is a class, and is a subclass of C. We emphasize here, that O and C' are objects of different characters.

In this thesis we use the notion of superclass in broader sense. Each class C' will be called superclass of the class C if C' is a superclass of some class which is superclass of C. If the original sense of superclass is needed, then we use a notion of *direct superclass*. All rules mentioned in this paragraph apply also to the notion of subclass.

The notion of a superclass makes it possible to introduce in Smalltalk a mechanism called *inheritance*. Generally speaking, inheritance means allowing objects of some class to use variables and methods from superclasses of this class. Of course, an object of the class C can execute all methods of the class C. Through inheritance it can perform methods of all superclasses of the class C. Inheritance of methods is limited by hiding. When adding a new method to some class we should make sure that its signature differs from signatures of all other methods in this class. However, it is possible to add to a class C a method of signature identical to a signature of method existing in a superclass of C. This new method hides methods from superclasses of C. If a method with a signature S was added to a class C, then by sending a message with a signature S to an object of the class C we invoke a method of the class C. Some exceptions are possible if a receiver of a message is described as *super*. Both inheritance and hiding of methods are introduced in Section 5.5.3 directly by giving a suitable rule.

Inheritance of methods is possible if variables are inherited. Suppose that a message sent to an object of a class C causes execution of a method M from the superclass C'. Names of object variables from objects of the class C' may occur in the method M. These names should be known, even if M is performed for a request sent to an object of the class C. This problem is solved by assuming that objects of the class C have also variables with identical names as objects of the class C'. Hence, we must assume that the function *Object_variable* satisfies the following condition for all $O, O' \in$ OBJECTS and $W \in$ NAMES:

if $Object\_variable\,(W, O')$ is defined and

$$Class\,(O') = \underbrace{Superclass\,(\ldots Superclass}_{k\ times}(Class\,(O))\ldots)$$

for some non-negative integer k, then $Object\_variable\,(W, O)$ is also defined.

A privilege of using class variables is inherited as well. It means that a method from a class C can read and change values of all class variables of superclasses of the class C. Like in the case of methods, this kind of inheritance is provided directly in Section 5.4.2 by giving a suitable rule.

In Smalltalk many class variables with the same name can exist, but for any class a name identifies a variable defined in this class. If a class C_1 is neither a subclass nor a superclass of the class C_2, then variables of the class C_1 may have the same name as variables of the class C_2. In the other cases, there are some limitations of naming variables, and Smalltalk does not allow to create a new variable with a name W in a class C if variable with the name W exists in a superclass of C. It suggests that if C_1 is a subclass of C_2, then names of class variables in the class C_1 should be different from such names in the class C_2. Then the function *Class_variable* should satisfy some additional conditions. But this rule is not fully complied. It may happen that there are variables with identical names in some class C and its superclass. Then the class variable from C hides variables of the same names existing in superclasses. In the rule SHARED_VARIABLE presented in Section 5.4.2 we give a solution of the ambiguity in our semantics.

In Smalltalk one also deals with indirect inheritance. Pool variables are inherited in such a way. A method of the class C can refer to pool variables only if these variables are in dictionaries declared in the class C. However, objects of the class C can use pool variables from dictionaries declared in a superclass of C executing methods from this superclass.

4.3 Representation of statements and expressions

Section 4.2 shows how objects and classes are represented in the algebra $\mathcal{A}_0$. In this section a representation of methods and their bodies is given. We consider here only ordinary methods (see Section 4.2.4), which are declared by their source code.

4.3.1 Grammar of Smalltalk

When defining an ordinary method (see Section 4.2.4), a programmer writes a text created in compliance with the syntax of Smalltalk which is given in [GR89, pp. 587–591]. For our purposes, we express this syntax in a form of the grammar $\mathcal{G}$ given in Appendix A. In this grammar, productions which allow replacing the following nonterminal symbols $\langle$*Unary_selector*$\rangle$, $\langle$*Binary_selector*$\rangle$, $\langle$*Variable_name*$\rangle$, $\langle$*variable_name*$\rangle$, $\langle$*Keyword*$\rangle$ and $\langle$*Literal*$\rangle$ are omitted. Symbols $\langle$*Unary_selector*$\rangle$, $\langle$*Variable_name*$\rangle$ and $\langle$*variable_name*$\rangle$ are substituted by words starting with a letter. There are names like *a*, *B* and *delta* or signatures of an unary method like *sqrt* and *neg*. From the symbol $\langle$*Binary_selector*$\rangle$ we can derive at most two-letters words created from special characters. Examples of such words are '$<=$', '$-$', '/' and '$*$'. Constants like 1 and *'abc'* are derived from $\langle$*Literal*$\rangle$. Fragments of *keyword* signatures may be derived from the symbol $\langle$*Keyword*$\rangle$. There are words with a trailing colon (e.g. *put:* and *value:*).

We use two different nonterminal symbols which look similar: $\langle$*Variable_name*$\rangle$ and $\langle$*variable_name*$\rangle$ to distinguish different variable occurrences. Both of them refer to variable names, but in different context: $\langle$*Variable_name*$\rangle$ denotes a variable name used as an expression. $\langle$*variable_name*$\rangle$ means a variable name used on the left side of assignment or in a variable declaration.

The grammar $\mathcal{G}$ of Smalltalk given in Appendix A is unambiguous. It means that any word derivable from a nonterminal symbol has the unique derivation tree. We leave this fact without a proof.

4.3.2 Expressions in Smalltalk

Every subword of a given word together with a position of its first letter is called a *fragment*. For instance, *'ababa'* has two fragments with a subword *'aba'*. There are the word *'aba'* starting from the first letter of *'ababa'* and the same word starting from the third letter. As long as it does not cause misunderstanding, fragments of words will be called subwords.

Every node in the derivation tree of a word W determines a subtree. This subtree generates a fragment of the word W, which consists of letters in leaves of this subtree. Figure 4.1 shows a part of the derivation tree of the method defined in Example 4.3. Nodes of this tree are labelled by nonterminal symbols. Parts that have been omitted are denoted by three dots. Notice that the node with the symbol $\langle$*Last_statement*$\rangle$ generates the fragment with the word *'↑((b neg - d)/(2*a))'*.

Suppose that there are given a word W and its derivation in the given grammar $\mathcal{G}$ from a nonterminal symbol. A nonempty fragment of W is called *subexpression*, if it is determined by a node of the derivation tree labelled by a nonterminal symbol belonging to the set of nonterminals:

$$\begin{aligned}
&\mathcal{N}_{Sub} = \{ \\
&\quad \langle \textit{Variable_name} \rangle, \langle \textit{Literal} \rangle, \langle \textit{Block} \rangle, \langle \textit{Unary_expression} \rangle, \langle \textit{Binary_expression} \rangle, \\
&\quad \langle \textit{Keyword_expression} \rangle, \langle \textit{Cascade_message_expression} \rangle \\
&\}
\end{aligned}$$

For instance, words *'((b neg - d)/(2*a))'*, *'((b*b) - 4*a*c) sqrt'* and *'(b*b) - 4*a*c'* are subexpressions of the method from Example 4.3. This method has also three subexpressions *'b'* and one *'d'*.

In Smalltalk, an expression is a word, which can be derived from the nonterminal $\langle$*Expression*$\rangle$. If a word W is derived in the grammar $\mathcal{G}$ from either $\langle$*Statement*$\rangle$ or $\langle$*Last_statement*$\rangle$, then W is called a statement.

4.3.3 Universe EXPRESSIONS

Now we are going to define the universe EXPRESSIONS and the set $\mathcal{F}_E$ of functions defined on EXPRESSIONS. $\mathcal{F}_E$ consists of the following functions:

$$\textit{Name} : \text{EXPRESSIONS} \rightarrow \text{NAMES},$$

$$\textit{Argument\_names} : \text{EXPRESSIONS} \rightarrow \text{LISTS\_OF\_NAMES},$$

$$\textit{Constant\_value} : \text{EXPRESSIONS} \rightarrow \text{OBJECTS},$$

$$\textit{First\_statement} : \text{EXPRESSIONS} \rightarrow \text{EXPRESSIONS},$$

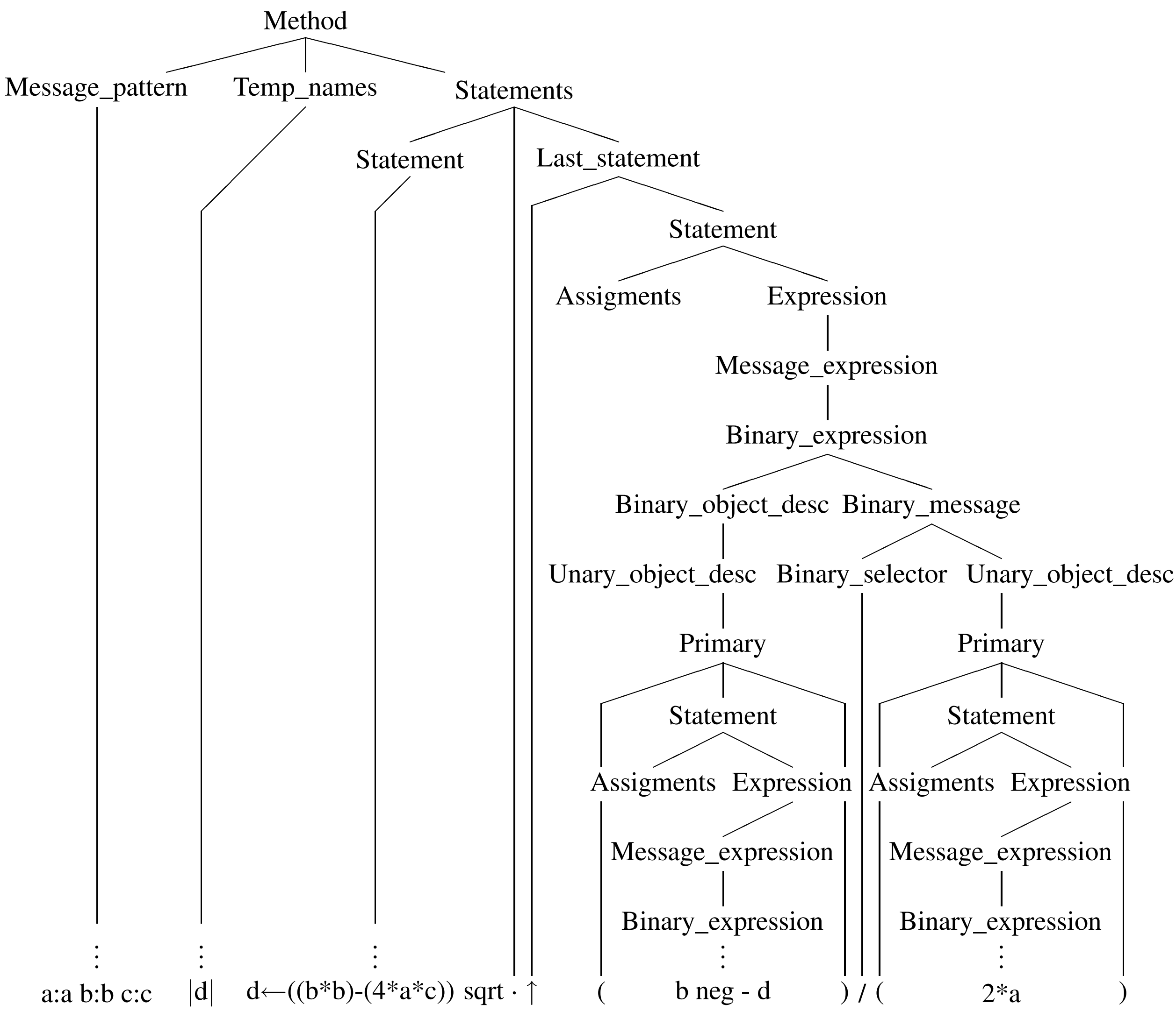

Figure 4.1: A fragment of the derivation tree of the method of Example 4.3

Receiver : EXPRESSIONS → EXPRESSIONS,

First_argument : EXPRESSIONS → EXPRESSIONS,

Next_argument : EXPRESSIONS → EXPRESSIONS,

Superexpression : EXPRESSIONS → EXPRESSIONS,

Assignments : EXPRESSIONS → LISTS_OF_NAMES,

Next_statement : EXPRESSIONS → EXPRESSIONS,

Return_prefix : EXPRESSIONS → NAMES.

For every method we consider a set of subexpressions of this method. We shall represent set of representations of subexpressions in such a way that different methods have disjoint representations. This assumption is necessary, because it is possible to declare two methods such that their sets of subexpressions consist of the same/identical element.

We define the set EXPRESSIONS as a set of elements which represent subexpressions. We assume that every element of EXPRESSIONS represents one subexpression of a method, and every subexpression of a method is represented by an element of EXPRESSIONS.

If $E \in$ EXPRESSIONS represents a subexpression of a method $\mathcal{M}$, then all functions belonging to $\mathcal{F}_E$ which have values in EXPRESSIONS map E onto some other elements which represent a subexpression of $\mathcal{M}$. Thus, to define a function in $\mathcal{F}_E$ we take the source of a method and define these functions for elements representing subexpressions of this method.

Let $\mathcal{M}$ be a method given in its source form. We assume that $\mathcal{M}$ was verified and accepted by the compiler of Smalltalk, and therefore

- the text of the method $\mathcal{M}$ can be derived in the Smalltalk grammar $\mathcal{G}$, (hence there exists the derivation tree of $\mathcal{M}$),
- neither argument names nor pseudovariable names occur on the left hand side of an assignment operator in $\mathcal{M}$
- all variable names occurring in $\mathcal{M}$ are names of admissible variables.

By *Exp* we we denote the set

$$
\begin{aligned}
&Exp = \{\\
&\quad \langle Statements\rangle, \langle Last\_statement\rangle, \langle Statement\rangle, \langle Expression\rangle, \langle Primary\rangle,\\
&\quad \langle Variable\_name\rangle, \langle Literal\rangle, \langle Unary\_object\_description\rangle, \langle Block\rangle,\\
&\quad \langle Unary\_expression\rangle, \langle Binary\_object\_description\rangle, \langle Binary\_expression\rangle,\\
&\quad \langle Keyword\_expression\rangle, \langle Message\_expression\rangle, \langle Cascade\_message\_expression\rangle\\
&\quad \}
\end{aligned}
$$

Suppose that $\mathcal{N}$ is a node of this tree labelled by an element of *Exp*. For every such node $\mathcal{N}$ we shall indicate an element $\mathcal{E} \in$ EXPRESSIONS corresponding to $\mathcal{N}$. The correspondence between nodes and elements of EXPRESSIONS will be defined by induction. Here each node labelled by a nonterminal in $\mathcal{N}_{Sub}$ we represent by an element of EXPRESSIONS. $\mathcal{E} \in$ EXPRESSIONS is defined for $\mathcal{W}$, then $\mathcal{E}$ is defined for $\mathcal{N}$.

Notions of expression and subexpression seem to be very similar. We explain the difference using an example. Consider a word $\mathcal{W}$ equal to *'(5)'* and its derivation tree $\mathcal{T}$ with the root labelled by $\langle Expression\rangle$. This tree can be represented by derivation of the form:

Derivation 4.1

$$
\langle Expression\rangle \Longrightarrow \langle Primary\rangle \Longrightarrow (\ \langle Statement\rangle\ ) \Longrightarrow (\ \langle Assignments\rangle\ \langle Expression\rangle\ ) \Longrightarrow
$$

$$
(\ \langle Expression\rangle\ ) \Longrightarrow (\ \langle Primary\rangle\ ) \Longrightarrow (\ \langle Literal\rangle\ ) \Longrightarrow (5)
$$

It is easy to notice, that $\mathcal{W}$ consists of two expressions: *'(5)'* and *'5'*, but only one subexpression *'5'*. These two expressions are representations of the same term and evaluate to the same object. So, the same element of the universe EXPRESSIONS should serve as a representation of these expressions.

We use the following conventions in the definition presented in this chapter: we say that an element $\mathsf{e} \in$ EXPRESSIONS *is defined* for some node of derivation tree labelled by nonterminals belonging to $\mathcal{N}_{Sub}$. We also say that an element $\mathsf{e} \in$ EXPRESSIONS *corresponds* to some node of derivation tree labelled by nonterminals belonging to *Exp*. It implies, that if e is defined for a node $\mathcal{N}$, then e corresponds to $\mathcal{N}$. The definition given in this chapter implies, that e is defined for the lowest node among the nodes for which it corresponds. From the point of view of semantics the nodes to which e corresponds and is not defined can be considered a syntactic sugar.

For Derivation 4.1 there exist an element $\mathsf{e} \in$ EXPRESSIONS which is defined for the node labelled by $\langle Literal \rangle$ of the derivation tree $\mathcal{T}$. This element e corresponds to all nodes of $\mathcal{T}$ labelled by $\langle Primary \rangle$, $\langle Statement \rangle$ and $\langle Expression \rangle$.

Below we give the definition of the universe EXPRESSIONS and associated functions for a method $\mathcal{M}$. We assume, that a derivation tree $\mathcal{T}$ of $\mathcal{M}$ is given. Instead of drawing fragments of $\mathcal{T}$ we will present corresponding derivations. The symbol $\Longrightarrow$ denotes one step of derivation, $\stackrel{*}{\Longrightarrow}$ denotes the reflexive and transitive closure of $\Longrightarrow$. Derivations are numbered. The definition is given by bottom–up induction over nodes of $\mathcal{T}$. Below we denote by $\mathcal{W}$ the word generated by a node $\mathcal{N}$.

4.3.3.1 Variables

Suppose that a node $\mathcal{N}$ is labelled by the symbol $\langle Variable\_name \rangle$. Then $\mathcal{W}$ is a name of a variable. For $\mathcal{N}$ we choose an element $\mathcal{E} \in$ EXPRESSIONS and we put

$$Name\,(\mathcal{E}) = \mathcal{W}.$$

4.3.3.2 Literal constants

Suppose that $\mathcal{N}$ is labelled by the symbol $\langle Literal \rangle$. Then $\mathcal{W}$ denotes a constant, like 123 or *"abc"*. Constants are objects, too. We assume, that for each such constant there exists a corresponding element $O \in$ OBJECTS. For the node $\mathcal{N}$ we choose an element $\mathcal{E} \in$ EXPRESSIONS, and we put

$$Constant\_value\,(\mathcal{E}) = O.$$

4.3.3.3 Block definitions

Let $\mathcal{N}$ be a node labelled by the symbol $\langle Block \rangle$. For such $\mathcal{N}$ we choose an element $\mathcal{E} \in$ EXPRESSIONS.

In the grammar $\mathcal{G}$, a derivation from $\langle Block \rangle$ has the following form:

Derivation 4.2

$$\langle Block \rangle \stackrel{*}{\Longrightarrow}$$
$$\text{`[' `:'}\langle variable\_name \rangle \ldots \text{ `:'}\langle variable\_name \rangle \text{ `|' } \langle Statements \rangle \text{ `]' } \stackrel{*}{\Longrightarrow}$$
$$\text{`[' `:'}W_1 \ldots \text{ `:'}W_k \text{ `|' W' `]'} = \mathcal{W}$$

where $W_1, \ldots, W_k$ are words derived from $\langle variable\_name \rangle$ for some $k \geq 0$, W' is derived from $\langle Statements \rangle$ (k may be equal to 0, and W' may be empty). In this case we put

$$Argument\_names\,(\mathcal{E}) = [W_1 \ \ldots \ W_k]$$

where $[W_1 \ \ldots \ W_k]$ denotes the list of $W_1, \ldots, W_k$ (see Chapter 4.2). If $\mathcal{W}$ is of the form $[\mid \ W' \ ]$, $[\ W' \ ]$ or $[\,]$, then we put

$$Argument\_names\,(\mathcal{E}) = [\,].$$

The function *First_statement* is defined on $\mathcal{E}$ if and only if $\mathcal{W}$ is different from $[\,]$. In this case there exists a node $\mathcal{N}'$ in the derivation tree of $\mathcal{M}$ labelled the symbol $\langle Statements \rangle$ determining the word W'. Let $\mathcal{E}' \in$ EXPRESSIONS be an element which corresponds to $\mathcal{N}'$. We put

$$First\_statement\,(\mathcal{E}) = \mathcal{E}'.$$

4.3.3.4 Some simple cases

Consider a node $\mathcal{N}$ with one of the following symbols: $\langle Expression \rangle$, $\langle Primary \rangle$, $\langle Unary\_object\_description \rangle$, $\langle Binary\_object\_description \rangle$ or $\langle Message\_expression \rangle$. It is easy to notice that $\mathcal{N}$ has in the derivation tree of $\mathcal{M}$ exactly one son $\mathcal{N}'$ labelled by a nonterminal. $\mathcal{N}'$ must be labelled by the nonterminal belonging to $\mathcal{N}_{Sub}$, i.e. $\mathcal{N}'$ denotes some subexpression. So by induction there exists $\mathcal{E}' \in$ EXPRESSIONS which corresponds to the node $\mathcal{N}'$, and $\mathcal{E}'$ corresponds to $\mathcal{N}$.

4.3.3.5 Message expressions

By $\langle expression \rangle$ we denote one of non–terminals: $\langle Unary\_expressions \rangle$, $\langle Binary\_expressions \rangle$, $\langle Keyword\_expressions \rangle$ or $\langle Cascade\_message\_expression \rangle$. We consider a node $\mathcal{N}$ labelled by a nonterminal $\langle expression \rangle$. Each such a node generates some kind of message expression which is also a subexpression of the method $\mathcal{M}$. For this $\mathcal{N}$ we define an element $\mathcal{E} \in$ EXPRESSIONS.

The word $\mathcal{W}$ generated by $\mathcal{N}$ expresses sending a request for execution of a method to some receiver. It has two parts. First one describes a receiver, the latter contains a name of a method and arguments. Below Derivation 4.3 corresponds to message expression and Derivation 4.4 corresponds to cascade message expression. Derivation of the word $\mathcal{W}$ from $\langle expression \rangle$ has either the form

Derivation 4.3

$$\langle expression \rangle \stackrel{*}{\Longrightarrow} \langle object\_description \rangle \langle message \rangle \Longrightarrow O\ M = \mathcal{W}$$

or the form

Derivation 4.4

$$\langle expression \rangle \stackrel{*}{\Longrightarrow} \langle object\_description \rangle `;' \langle message \rangle \stackrel{*}{\Longrightarrow} O`;'M = \mathcal{W}$$

where

- $\langle object\_description \rangle$ is one of the following nonterminal: $\langle Unary\_object\_description \rangle$, $\langle Binary\_object\_description \rangle$, $\langle Unary\_expression \rangle$, $\langle Binary\_expression \rangle$, $\langle Keyword\_expression \rangle$ or $\langle Cascade\_message\_expression \rangle$,
- $\langle message \rangle$ is one of the following nonterminal: $\langle Unary\_message \rangle$, $\langle Binary\_message \rangle$ or $\langle Keyword\_message \rangle$,
- words O and M are derived from $\langle object\_description \rangle$ and $\langle message \rangle$, respectively.

Consequently, in the derivation tree of $\mathcal{M}$ there are two nodes: $\mathcal{N}_0$ with the nonterminal $\langle object\_description \rangle$, determining the word O, and $\mathcal{N}'$ labelled by $\langle message \rangle$, determining the word M.

By induction, there exists $\mathcal{E}_0 \in$ EXPRESSIONS which corresponds to $\mathcal{N}_0$. Thus, $\mathcal{E}_0$ represents a receiver of the expression $\mathcal{W}$.

The relationship between $\mathcal{E}$ and $\mathcal{E}_0$ depends on the symbol $\langle expression \rangle$. We put

$$Receiver\,(\mathcal{E}) = Receiver\,(\mathcal{E}_0),$$

if $\langle expression \rangle$ is equal to $\langle Cascade\_message\_expression \rangle$, and we put

$$Receiver\,(\mathcal{E}) = \mathcal{E}_0$$

in other cases. In both cases, we put

$$Superexpression\,(\mathcal{E}_0) = \mathcal{E}.$$

This intuitively means, that computations of the value of $\mathcal{E}$ are performed after computing the value of $\mathcal{E}_0$.

Values of other functions defined for $\mathcal{E}$ are determined by the node which is the second son of $\mathcal{N}$, and determines the word M.

If $\langle message \rangle$ is $\langle Keyword\_message \rangle$, then a derivation of M from $\langle message \rangle$ has the form:

Derivation 4.5

$$\langle Keyword\_message \rangle \Longrightarrow$$

$$\langle Keyword \rangle \langle Binary\_object\_description \rangle \ldots \langle Keyword \rangle \langle Binary\_object\_description \rangle \stackrel{*}{\Longrightarrow}$$

$$M$$

Hence,

$$M = W_1 O_1 \dots W_n O_n$$

for words $O_1, \dots, O_n$ derived from $\langle$*Binary_object_description*$\rangle$ and $W_1, \dots, W_n$ derived from $\langle$*Keyword*$\rangle$. In Smalltalk, the word M is considered as a request for execution of a method of a signature $W_1 \dots W_n$ with arguments described by words $O_1, \dots, O_n$.

The form of a derivation of M implies that in the derivation tree of $\mathcal{M}$ there exist nodes $\mathcal{N}_1, \dots, \mathcal{N}_n$ with labels $\langle$*Binary_object_description*$\rangle$, determining words $O_1, \dots, O_n$, respectively. By induction hypothesis, there exist $\mathcal{E}_1, \dots, \mathcal{E}_n$ which correspond to $\mathcal{N}_1, \dots, \mathcal{N}_n$, respectively.

For $\mathcal{E}$ and $\mathcal{E}_1, \dots, \mathcal{E}_n$ we put

$$\mathit{Name}\,(\mathcal{E}) = W_1 \dots W_n,$$

$$\mathit{First\_argument}\,(\mathcal{E}) = \mathcal{E}_1,$$

$$\mathit{Next\_argument}\,(\mathcal{E}_i) = \mathcal{E}_{i+1}$$

for $i = 1, \dots, n-1$, and

$$\mathit{Superexpression}\,(\mathcal{E}_n) = \mathcal{E}.$$

If $E \in \textsc{Expressions}$ represents a request for execution of a method, then using these functions, we can fix a name of a method to be executed and elements which represent arguments. After computation of values of arguments, we can go back to E.

If $\mathcal{N}'$ is a node labelled by $\langle$*Binary_message*$\rangle$, the derivation of M from $\langle$*Binary_message*$\rangle$ has the following form

Derivation 4.6

$$\langle \mathit{Binary\_message} \rangle \Longrightarrow \langle \mathit{Binary\_selector} \rangle \langle \mathit{Unary\_object\_description} \rangle \stackrel{*}{\Longrightarrow} W_1 O_1 = M$$

for some words W_1 and O_1 derived from symbols $\langle$*Binary_selector*$\rangle$ and $\langle$*Unary_object_description*$\rangle$, respectively. There exists a node $\mathcal{N}_1$ in the derivation tree of $\mathcal{M}$ determining a word O_1, and $\mathcal{E}_1 \in \textsc{Expressions}$ which corresponds to $\mathcal{N}_1$. We put:

$$\mathit{Name}\,(\mathcal{E}) = W_1,$$

$$\mathit{First\_argument}\,(\mathcal{E}) = \mathcal{E}_1,$$

$$\mathit{Superexpression}\,(\mathcal{E}_1) = \mathcal{E}.$$

If the node $\mathcal{N}'$ is labelled by $\langle$*Unary_message*$\rangle$, then the derivation of M has the form

Derivation 4.7

$$\langle \mathit{Unary\_message} \rangle \Longrightarrow \langle \mathit{Unary\_selector} \rangle \stackrel{*}{\Longrightarrow} M$$

In this case the message M consists of a name of a method only, and we add

$$\mathit{Name}\,(\mathcal{E}) = M$$

to the definition of $\mathcal{A}_0$.

4.3.3.6 Example of block

We illustrate our definition by an expression

```
[ :x :y | expr ] value: a value: b
```

A derivation tree of this expression is shown in Figure 4.2. Nodes of this tree, which generate subexpressions are underlined.

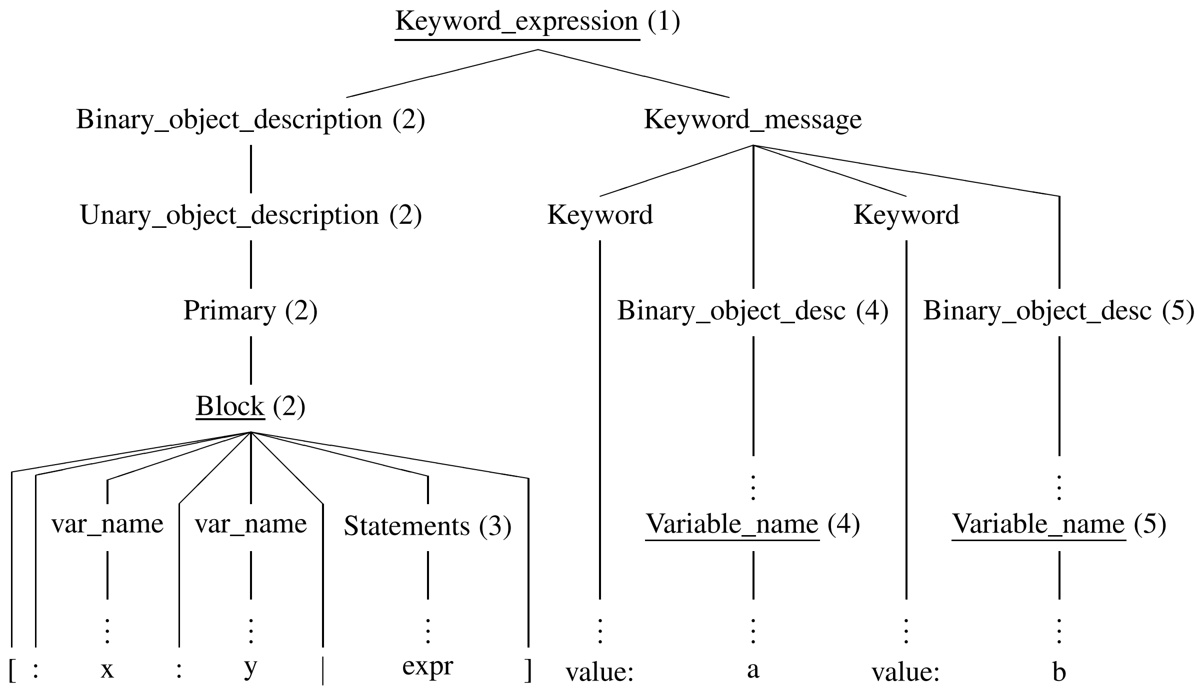

Figure 4.2: The derivation tree of an expression `[ :x :y | expr ] value: a value: b`

In Figure 4.3 elements of EXPRESSIONS are denoted by circles and functions are denoted by arcs representing the expression `'[ :x :y | expr ] value: a value: b'`.

Some nodes in Figure 4.2 are additionally labelled by a positive integers to identify them. These numbers correspond to numbers of nodes in Figure 4.3.

4.3.3.7 Example of cascade message

An expression

```
a sqrt - 1; sqrt.
```

serves as another example. It is a cascade meessage, because two methods are sent to the value of the expression `'a sqrt'`: binary method `-` with argument `1` and unary method `sqrt`.

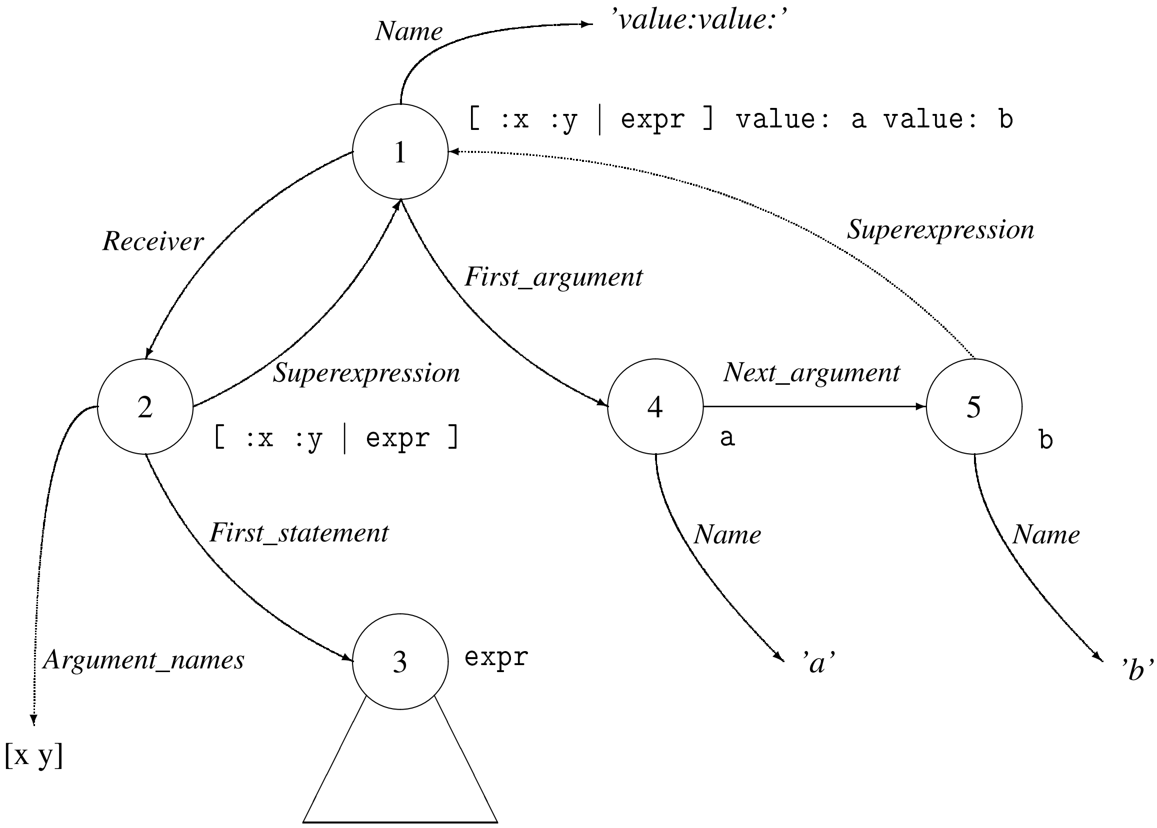

Figure 4.3: A block and a keyword expression

4.3.3.8 Statements

Now we shall consider nodes $\mathcal{N}$ labelled by $\langle Statement\rangle$. The derivation of the word from $\langle Statement\rangle$ has the form

Derivation 4.8

$$\langle Statement\rangle \Longrightarrow \langle Assignments\rangle\langle Expression\rangle \stackrel{*}{\Longrightarrow} W_1 \text{ `}\leftarrow\text{ ' } \ldots W_n \text{ `}\leftarrow\text{ ' } W' = \mathcal{W}$$

for word W' which is derived from $\langle Expression\rangle$. If $\mathcal{N}$ determines $\mathcal{W}$ in the derivation tree of $\mathcal{M}$, then there exists a node $\mathcal{N}'$ labelled by $\langle Expression\rangle$ determining the word W'. By induction there exists an element $\mathcal{E}'$ of EXPRESSIONS which corresponds to $\mathcal{N}'$. We define that $\mathcal{E}'$ also corresponds to $\mathcal{N}$.

During computation, the value of expression W' is assigned to variables of names $W_1, \ldots, W_n$. These names are stored in the list *Assignments* $(\mathcal{E}')$. But it is possible, that this list consist of other variable names. Suppose that W' is a word of the form: `'(i ← (j ← (W'')))'`. Then `i` and `j` should also be members of the list *Assignments* $(\mathcal{E}')$.

Therefore the definition of the function *Assignments* is following: let $\mathcal{E}'$ be an element of EXPRESSIONS corresponding to some node labelled by $\langle Statement\rangle$. $\mathcal{E}'$ can also corresponds

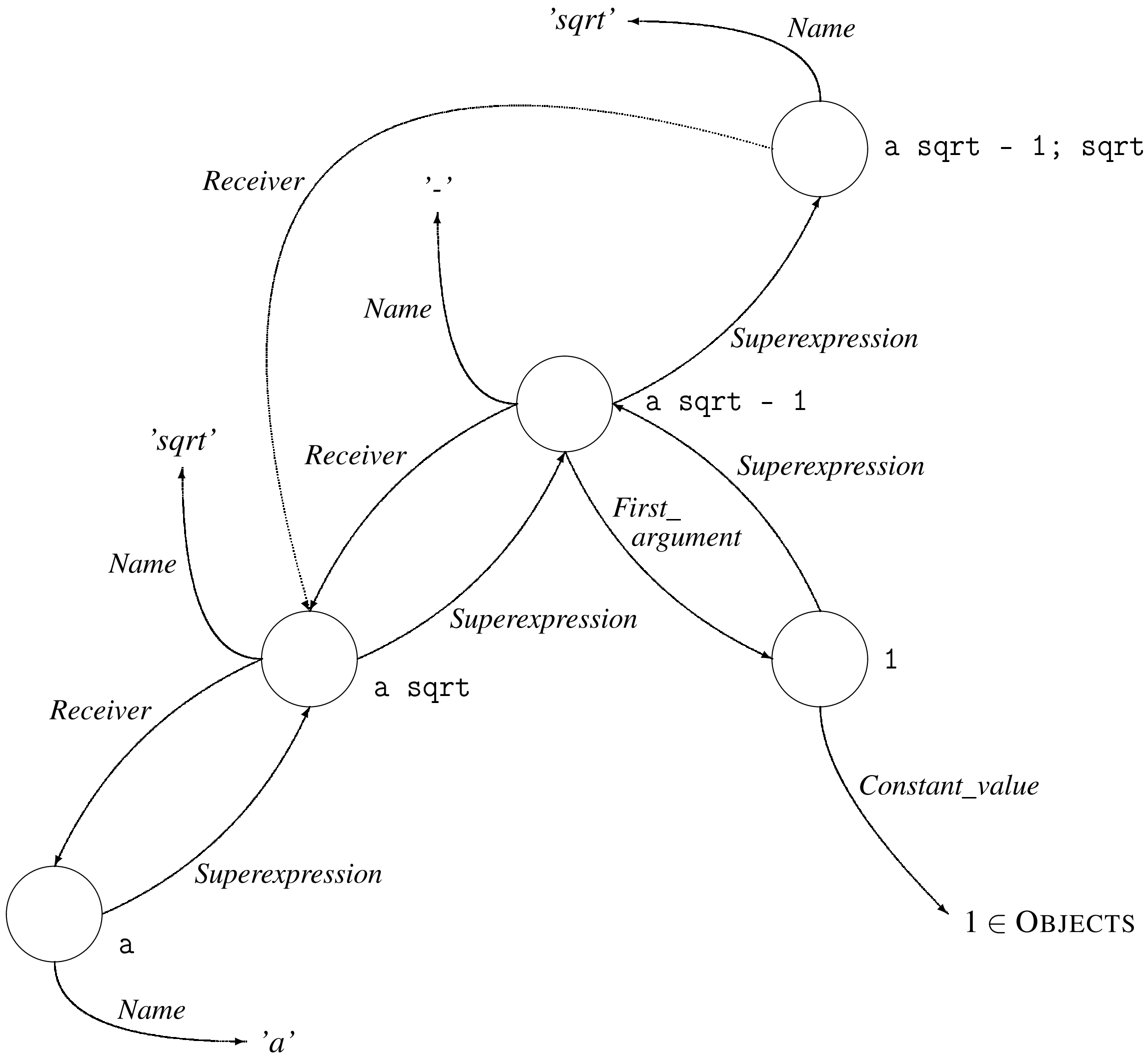

Figure 4.4: Cascade message, and unary, and binary expressions

to other nodes labelled by ⟨*Statement*⟩. Let $E_{\mathcal{E}'}$ be a set of all such nodes. For each $\mathcal{R} \in E_{\mathcal{E}'}$ we define a list $\mathcal{L}_{\mathcal{R}}$ which includes all variable names derived from ⟨*Assignments*⟩ for $\mathcal{R}$. The value of *Assignments* $(\mathcal{E}')$ is a concatenation of the lists $\mathcal{L}_{\mathcal{R}}$, where $\mathcal{R} \in E_{\mathcal{E}'}$.

This part of the definition of EXPRESSIONS and next are illustrated in Figure 4.5.

4.3.3.9 Last statements

Suppose that a node $\mathcal{N}$ is labelled by ⟨*Last_statement*⟩. In the derivation tree, such a node has always exactly one son $\mathcal{N}'$ with the symbol ⟨*Statement*⟩. By induction, there exists $\mathcal{E}' \in$ EXPRESSIONS which corresponds to $\mathcal{N}'$. We define, that $\mathcal{E}'$ corresponds to $\mathcal{N}$, too. If $\mathcal{N}$ has a son with the character ↑, then we put

$$Return\_prefix\,(\mathcal{E}') = \uparrow$$

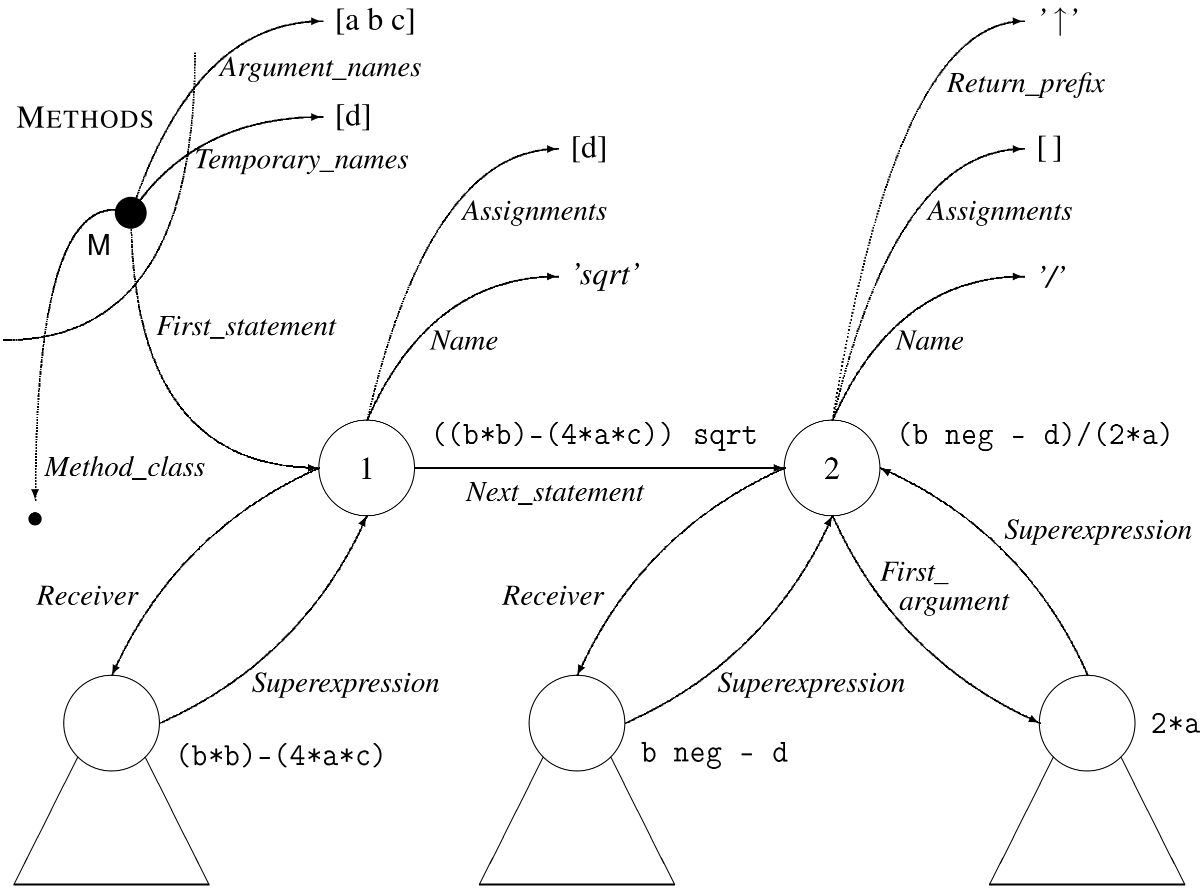

Figure 4.5: A sequence of statements of the example 4.3

otherwise we leave

$$Return\_prefix\,(\mathcal{E}') = \mathsf{undef}$$

4.3.3.10 Sequences of statements

It follows from the form of the grammar $\mathcal{G}$ that if the node $\mathcal{N}$ with the nonterminal $\langle Statements \rangle$ generates a word $\mathcal{W}$, then

$$W = W_1\text{‘.’}W_2\text{‘.’}\ \dots\ \text{‘.’}W_n$$

for some words W_i which are derived from symbols $\langle Statement \rangle$ or $\langle Last\_statement \rangle$. There exist nodes $\mathcal{N}_1, \dots, \mathcal{N}_n$ with symbols $\langle Statement \rangle$ or $\langle Last\_statement \rangle$ in the derivation tree of $\mathcal{M}$ generating words $W_1, W_2, \dots, W_n$. Let $\mathcal{E}_1, \mathcal{E}_2, \dots, \mathcal{E}_n$ be elements of EXPRESSIONS which correspond to these nodes. We define that $\mathcal{E}_1$ corresponds to $\mathcal{N}$, and we put

$$Next\_statement\,(\mathcal{E}_i) = \mathcal{E}_{i+1}$$

for all $i = 1, \dots, n-1$.

4.3.4 Functions defined on the universe METHODS

In previous sections, we have defined the set EXPRESSIONS together with functions defined on this set. Each element of EXPRESSIONS represents an expression in a method. In $\mathcal{A}_0$ methods are represented by elements of the universe METHODS. In this section we define functions *First_statement* and *Temporary_names* defined on METHODS.

In the derivation tree of each non-empty method the root node has a son with the nonterminal $\langle Statements \rangle$. Let $\mathsf{M} \in$ METHODS represent a method $\mathcal{M}$. In the derivation tree of $\mathcal{M}$ the root node has a son $\mathcal{N}$ with $\langle Statements \rangle$. Let $\mathcal{E}$ be an element of EXPRESSIONS which corresponds to $\mathcal{N}$. We put

$$First\_statement\,(\mathsf{M}) = \mathcal{E}.$$

Now, we give a definition of the function *Temporary_names* for M. The definition of *Temporary_names*(M) depends on explicit declaration of temporary variables, and on declarations of arguments of blocks included in $\mathcal{M}$. If the derivation tree $\mathcal{T}$ of $\mathcal{M}$ contains a node with the symbol $\langle Temporary\_names \rangle$, then this node generates a word which consists of words $W_1, \ldots, W_n$ between characters '|'. Suppose that $\mathcal{E}_1, \ldots, \mathcal{E}_m \in$ EXPRESSIONS correspond to nodes of $\mathcal{T}$ labelled by $\langle Block \rangle$. For each $\mathcal{E}_1, \ldots, \mathcal{E}_m$, the function *Argument_names* is defined (see Section 4.3.3.3). We define *Temporary_names* (M) as the concatenation of a list which contains words $W_1, \ldots, W_n$, and all lists *Argument_names* $(\mathcal{E}_1), \ldots,$ *Argument_names* $(\mathcal{E}_m)$. If the derivation tree $\mathcal{T}$ of $\mathcal{M}$ does not contains any node with the symbol $\langle Temporary\_names \rangle$, then *Temporary_names* contains only variable names included in *Argument_names* $(\mathcal{E}_1), \ldots,$ *Argument_names* $(\mathcal{E}_m)$.

If $\mathcal{M}$ is the method with the empty body, then the derivation of this method in the grammar $\mathcal{G}$ has the form

Derivation 4.9

$$\langle Method \rangle \Longrightarrow \langle Message\_pattern \rangle \stackrel{*}{\Longrightarrow} \mathcal{M}$$

For such M we define only the value of *Temporary_names*. We put

$$Temporary\_names\,(\mathsf{M}) = [\,]$$

In Figure 4.5 we give a representation of the method from Example 4.3.

4.4 Properties of $\mathcal{A}_0$

In this section we prove properties of the part of algebra we have defined so far.

We use here an auxiliary function *Gen*, which for a given node $\mathcal{N}$ of the derivation tree of a method $\mathcal{M}$, gives the subword of $\mathcal{M}$ generated from $\mathcal{N}$.

Lemma 4.1 *Let $\mathcal{N}$ be a node of the derivation tree labelled by a nonterminal form the set $Exp/\{\langle Statements\rangle\}$, $\mathcal{E}$ be the element of* EXPRESSIONS *which corresponds to $\mathcal{N}$, and let $\mathcal{N}'$ be a node for which $\mathcal{E}$ is defined*[2]. *Then*

$$Gen(\mathcal{N}) = S_1 Gen(\mathcal{N}')S_2$$

where S_2 is equal to '$)$'*.

Proof
The proof follows from the fact, that for each nonterminal $\langle exp\rangle \in Exp/(\mathcal{N}_{Sub} \cup \{\langle Statements\rangle\})$ derivation rule of the grammar $\mathcal{G}$ has the form

$$\langle exp\rangle ::= S\ \langle exp'\rangle\ ')'$$

where $\langle exp'\rangle$ belongs to Exp, and S consists of '(', '$\leftarrow$ ', and variable names.

■

Lemma 4.2 *Let e be any element of the universe* EXPRESSIONS. *Then at most one of the functions Next_statement, Next_argument and Superexpression is defined on e.*

Proof
We use the fact that functions *Next_statement*, *Next_argument* and *Superexpression* are defined when certain productions are used. If two different functions are defined for the same element *e*, then we can consider nodes which corresponds to *e* and words generated from these nodes. We show, that these words have properties which contradict to each other.

Let *e* be fixed. We assume here that *e* is defined for a node $\mathcal{N}$ of derivation tree, and $\mathcal{N}$ is labelled by a nonterminal $\langle exp\rangle$, where $\langle exp\rangle$ belongs to the set $\mathcal{N}_{Sub}$ defined in Section 4.3.2.

If the function *Next_statement* is defined, then in the derivation of a method there exists a derivation

Derivation 4.10
$$\langle Statement\rangle\text{'.'}\ \langle Statement\rangle \stackrel{*}{\Longrightarrow} S_1\ \langle exp\rangle\ S_2\text{'.'}\ \langle Statement\rangle \stackrel{*}{\Longrightarrow} S_1\ \mathrm{Gen}(\mathcal{N})\ S_2\text{'.'}\ \langle Statement\rangle$$

If *Next_argument* is defined, then there exists a derivation of the form

Derivation 4.11
$$\langle Binary\_object\_description\rangle\langle Keyword\rangle\langle Binary\_object\_description\rangle \stackrel{*}{\Longrightarrow}$$
$$S_1'\ \langle exp\rangle\ S_2'\ \langle Keyword\rangle\ \langle Binary\_object\_description\rangle \stackrel{*}{\Longrightarrow}$$
$$S_1'\ \mathrm{Gen}(\mathcal{N})\ S_2'\ \langle Keyword\rangle\ \langle Binary\_object\_description\rangle$$

[2] Recall that there may be many nodes $\mathcal{N}$ which are ancestor to $\mathcal{N}'$ which correspond to $\mathcal{E}$.

(see Derivation 4.5). By Lemma 4.1 S_2 and S'_2 consist only right brackets. In both derivations Gen($\mathcal{N}$) denotes the same fragment of a method, hence words following Gen($\mathcal{N}$) must be the same in both derivations. But it does not hold, therefore *Next_statement* and *Next_argument* can not be both defined for e.

Suppose that *Superexpression* is defined on e. This means that either e corresponds to the receiver of a message expression, or is the last argument of a message expression. In the first case there exists a derivation of the form:

Derivation 4.12

$$\langle message\rangle \stackrel{*}{\Longrightarrow} \langle object\_description\rangle\ \langle msg\rangle \stackrel{*}{\Longrightarrow} T_1\ \langle exp\rangle\ T_2\ \langle msg\rangle \stackrel{*}{\Longrightarrow}$$

$$T'_1\ \mathrm{Gen}(\mathcal{N})\ T'_2\ \langle msg\rangle \stackrel{*}{\Longrightarrow} T'_1\ \mathrm{Gen}(\mathcal{N})\ T'_2\ m\ m'$$

where m is a word over terminal alphabet, and $\langle message\rangle$ is either $\langle Unary\_expression\rangle$, or $\langle Binary\_expression\rangle$, or $\langle Keyword\_expression\rangle$. The word m is either a message, in case of unary or binary message, or a fragment of keyword message. The word m' can be the empty word if $\langle message\rangle$ is $\langle Unary\_expression\rangle$. Lemma 4.1 implies, that T'_2 consists of right brackets. Therefore *Next_statement* can not be defined on e. Now, by the analysis of the construction given in Section 4.3.3.5 implies, that the function *Next_arguments* can not be defined on e.

The case, where the function *Superexpression* is defined on e and is the last argument of a message expression, is more complicated. Let e' be such an element that

$$Superexpression(e) = e'$$

and $\mathcal{N}'$ is a node of the derivation tree for which e' is defined. Let $\langle exp'\rangle$ be a label of $\mathcal{N}'$. There are two possible cases: $\langle exp'\rangle$ appears in the derivation from $\langle Statement\rangle$ or $\langle Statement\rangle$ appears in derivation from $\langle exp'\rangle$. In the first case we obtain the derivation

Derivation 4.13

$$\langle Statement\rangle \stackrel{*}{\Longrightarrow} S_1\ \langle exp'\rangle\ S_2 \stackrel{*}{\Longrightarrow} S_1\ S'_1\ \langle exp\rangle\ S'_2\ \langle msg\rangle\ S_2$$

where $\langle exp'\rangle$ is one of the nonterminals $\langle Unary\_expression\rangle$, $\langle Binary\_expression\rangle$ or $\langle Keyword\_expression\rangle$ from which some message expression is derived, and $\langle exp\rangle$ is a receiver of the message. But in this case e is not assigned to a node labelled by $\langle Statement\rangle$, therefore $\langle exp'\rangle$ can not appear in derivation from $\langle Statement\rangle$. In the second case we obtain the derivation

Derivation 4.14

$$\langle exp'\rangle \stackrel{*}{\Longrightarrow} S_1\ \langle Statements\rangle\ S_2 \stackrel{*}{\Longrightarrow}$$

$$S_1\ \langle Statement\rangle\text{'.'}\ \ldots\ \langle Statement\rangle\text{'.'}\ \ldots\ \langle Last\_statement\rangle\ S_2 \stackrel{*}{\Longrightarrow}$$

$$S_1\ \langle Statement\rangle\text{'.'}\ \ldots\ S'_1\ \langle exp\rangle\ S'_2\text{'.'}\ \ldots\ \langle Last\_statement\rangle\ S_2$$

By analysis of the grammar $\mathcal{G}$, the word $S_1\langle Statement\rangle$'.' … $\langle Statement\rangle$'.' … $\langle Last\_statement\rangle S_2$ can be generated only from a node $\mathcal{N}'$ labelled by $\langle Block\rangle$. But in such a case, an element $e'' \in$ EXPRESSIONS is defined for $\mathcal{N}'$. Therefore *Superexpression* (e) can not be equal to e'.

■

4.5 States

4.5.1 Initial information

Each execution of a method $\mathcal{M}$ consists of sequential evaluation of Smalltalk expressions in some prescribed order. Some expressions may have the form of a request sent to an object for execution of a method. The evaluation of such an expression invokes a suitable method $\mathcal{M}$'. It causes that $\mathcal{M}$ becomes suspended and an execution of $\mathcal{M}$' begins. When execution of $\mathcal{M}$' is completed computations of $\mathcal{M}$ is reactivated. Therefore in Smalltalk all informations about suspended method are stored in a memory. Since the invoked method can invoke another one, during computations Smalltalk builds a sequence of informations describing activated methods.

In Smalltalk-80, objects of the class **MethodContext** are created and when the system starts to execute an invoked method it stores informations about methods performed so far. Each object of **MethodContext** contains a method in a compiled form, an object which has been requested for execution of this method, values of arguments and values of temporary variables. The objects of **MethodContext** allow also to find a method which created them, and remember the configuration they have been created. The definition of the universe STATES we give here is based on the model of the class **MethodContext**.

4.5.2 Universe STATES

To give a semantics of methods we define a universe STATES. At the beginning of computation this universe in the algebra $\mathcal{A}_0$ is empty. Therefore functions defined on STATES cannot be defined for any argument and values of constants declared as belonging to STATES are equal to undef. During the transformation of $\mathcal{A}_0$, the universe STATES is extended by new elements. Then values of functions on STATES are defined on these elements and values are assigned to constants with values from STATES. All details concerning STATES are defined by the program $\mathcal{R}_0$ given in Section 5.2 and following. Now, we shall only list all functions defined on STATES and constants with values in this universe, and describe their role in Smalltalk.

In the signature of algebra $\mathcal{A}_0$ there are three functions defined on STATES, which describe the method which is executed in a given state:

$$\mathit{Current\_expression} : \text{STATES} \to \text{EXPRESSIONS},$$

$$\mathit{Method\_class} : \text{STATES} \to \text{CLASSES},$$

$$\mathit{Receiver} : \text{STATES} \to \text{OBJECTS}.$$

Suppose that $S \in$ STATES is created when a method M is activated by a request sent to an object O. *Current_expression* maps S into an element of EXPRESSIONS representing an expression of M, which is currently evaluated. When S is created, *Current_expression* (S) finds the body of M. Later it points to a part of M not yet performed. The function *Method_class* gives the class of the method activated in a given state. The value *Method_class* (S) represents a class in which M is defined. The function *Receiver* (defined on STATES) assigns the element of OBJECTS representing O to S. All of these functions are total.

The function

$$Temporary\_variable : \text{NAMES} \times \text{STATES} \to \text{VARIABLES}$$

from the algebra $\mathcal{A}_0$ was already mentioned in Section 4.2.3. Execution of a method needs creating and initializing temporary variables. For each pair (N,S), where S is a state and N is a name of a local variable or an argument of a method, the function *Temporary_variable* assigns element of VARIABLES.

Results of computing expressions of the method M in the state S are values of a function on pairs of expressions and states

$$Expression\_value : \text{EXPRESSIONS} \times \text{STATES} \to \text{OBJECTS}.$$

If

$$Expression\_value\,(E,S) = \mathsf{undef}$$

for some expression E, then the expression E is not yet evaluated in the state S, or was evaluated and then used in a later computations computations. If the value $Expression\_value\,(E,S)$ is defined, then it is an element of OBJECTS, representing the value of the expression E computed in this state.

Execution of a method generates a sequence of objects carrying informations about activated and suspended methods. During the transformation of $\mathcal{A}_0$ a sequence of elements of STATES playing a similar role is created. This sequence can be computed using a constant

$$\mathsf{Current\_state} \in \text{STATES}$$

and a function

$$Previous\_state : \text{STATES} \to \text{STATES}.$$

The constant Current_state points to the state which corresponds to the method most recently invoked. Moreover, for each S in this sequence, if the element preceding S exists, then it is equal to $Previous\_state\,(S)$.

In the algebra $\mathcal{A}_0$ we have also a total function

$$Home\_state : \text{STATES} \to \text{STATES}.$$

The equation $Home\_state\,(S) = S$ holds for each $S \in \text{STATES}$ created for executing a method. Additional details about this function are given in Section 4.5.3.

4.5.3 Blocks

In Smalltalk the programmer can use methods and expression derived from the nonterminal $\langle\mathsf{Block}\rangle$. Such an expression is a definition of a block. In Smalltalk every definition of a block acts similarly as a literal and describes an object called a block. This object stores a fragment of a method, which can be skipped or repeatedly executed, or is stored only and will be executed later. The difference between a literal and a definition of a block is that every execution of a method containing this definition creates a new block.

It is necessary to distinguish elements representing objects which are blocks. We assume that these elements belong to the set BLOCKS. Hence, it is obvious that

$$\text{BLOCKS} \subseteq \text{OBJECTS}.$$

All blocks are objects of the same class. We put in $\mathcal{A}_0$ a constant

$\mathsf{block} \in \text{CLASSES},$

Of course, if $B \in \text{BLOCKS}$ then $Class(B) = \mathsf{block}$ and $B \in \text{OBJECTS}$.

The definition of a block has the form $[\,: x_1 \ldots : x_n \mid s_1. \ \ldots \ .s_m\,]$, where $x_1, \ldots, x_n$ are names of block arguments and $s_1, \ldots, s_m$ are statements. The sequence $s_1. \ \ldots \ .s_m$ (consisting of words separated by a dot) is called the body of a block. The body of a block defined in the method M determines a part of the subset of EXPRESSIONS corresponding to the method M. First of all, blocks collect all informations contained in their definitions. The function

$First\_statement : \text{BLOCKS} \to \text{EXPRESSIONS}$

maps a block B onto an element of EXPRESSIONS which represents the first statement of the body of B. For a given block B, the function

$Argument\_names : \text{BLOCKS} \to \text{LISTS\_OF\_NAMES}$

returns a list of argument names which are given in the definition of B. Besides informations given in the definition, the block B holds the state S in which it was created. This state is a value of a function

$Home\_state : \text{BLOCKS} \to \text{STATES},$

for B. In this way it is possible to interpret names of variables and methods in the body of a block in the same way as at the moment when this block was created.

Names of the functions defined on BLOCKS are the same as functions defined on other universes. We can distinguish functions with the same names by examining either their arguments or domains.

The body of a block is computed by special primitive methods. In Smalltalk, these methods are named *'value'*, *'value:'* and *'value:value:'*. We assume that these methods are represented in $\mathcal{A}_0$ by elements of the universe METHODS, and are values of terms

$$Method\,('value', \mathsf{block}),$$
$$Method\,('value:', \mathsf{block}),$$
$$Method\,('value:value:', \mathsf{block})$$

The set of such values will be denoted by BLOCK_METHODS, so

$$\text{BLOCK\_METHODS} = \{ Method\,('value', \mathsf{block}),\ Method\,('value:', \mathsf{block}),\ Method\,('value:value:', \mathsf{block}), \ldots \}.$$

We shall write briefly that 'a block is executed' instead of 'a method executing the body of a block is executed'.

The methods executing blocks belong to the class Block. Hence, we assume that

$$Method\_class\,(M) = \mathsf{block}$$

for each element $M \in$ BLOCK_METHODS. Values of the function *Temporary_names* are not needed, so for each $M \in$ BLOCK_METHODS we put

$$Temporary\_names\,(M) = \mathsf{undef}.$$

The execution of a block is similar to the execution of a method. Each method and block is executed in a separated context, called *state* in our semantics. States are represented by elements of the universe STATES. There is one important difference between states created for execution of a block and of an ordinary method. In a state S created for execution of a block B, new local variables are not created, but local variables created in other state S' are used instead. The state S' is the value of the function *Home_state* on the argument B. The name of this function is the same as the function *Home_state* defined in Section 4.5.2. Both these functions will be used in such a way that the condition

$$Home\_state\,(S) = Home\_state\,(B)$$

will hold if the block B will be executed in the state S. The state in which B was created is the value of terms $Home\_state\,(B)$ and $Home\_state\,(S)$.

4.5.4 Auxiliary constants

In addition to function and constant symbols defined so far the signature of the algebra $\mathcal{A}_0$ contains the following constants:

Superfluous_state $\in$ STATES,

Assignments $\in$ LISTS_OF_NAMES,

Class $\in$ CLASSES,

Return_prefix $\in$ NAMES,

Last_value $\in$ OBJECTS,

Argument $\in$ EXPRESSIONS,

Argument_names $\in$ LISTS_OF_NAMES,

Temporary_names $\in$ LISTS_OF_NAMES,

Variable $\in$ VARIABLES,

Variable_name $\in$ VARIABLES,

and

Pool_names $\in$ LISTS_OF_NAMES.

These constants play only auxiliary role. Their meaning and interpretation is given in explanation of a transition rule $\mathcal{R}_0$.

4.5.5 The beginning of a computation

In this section we explain how to begin a computation. In Smalltalk, if a global variable *Application* is defined, and an object, which is the value of *Application* can perform a method *run*, a user can write an expression like

Example 4.4

```
Application run
```

The computation begins by running the initial method run on an initial object *Application*. An object O, which is the value of variable *Application* plays a role of initial object and *run* is an initial method. The initial expression can contain more complicated expressions (e.g. keyword message), but in order to simplify considerations given below we assume that computation starts from such a simple expression.

We define a beginning of computation in our algebra $\mathcal{A}_0$ simulating such initial object and method. We assume that initial object and its method are values of the constants

$\mathsf{Method} \in \textsc{Methods}$

and

$\mathsf{Receiver} \in \textsc{Objects}$

respectively.

Values of the constants Method and Receiver should be given by a user and we do not define their values. These constants should satisfy some conditions. For simplicity we assume, that initial expression contains a message expression with unary message.

Suppose that $M \in \textsc{Methods}$ and $O \in \textsc{Objects}$ are values of the constants Method and Receiver. Since we have assumed that initial expression contains a message expression with unary message, it follows that

$$Argument\_names\,(M) = [\,]$$

holds.

M and O can not be arbitrary. M should be a method defined for O, i.e. it should satisfy the condition:

$$Method\_class\,(M) = \underbrace{Superclass\,(\ldots Superclass}_{k\ times}(Class\,(O))\ldots)$$

Additionally, we assume that if $Method\_class\,(M) = \mathsf{Block}$, then $Class\,(O) = \mathsf{Block}$. If O and M do not satisfy this condition, then it leads to incorrect computation: we try to compute a method, which is not defined for the receiver.

If computation of Smalltalk program ends, then a value (an object) of the initial expression appears as a result of computation. At the end of the evaluation of the algebra the value of the constant Last_value corresponds to the object computed by a real Smalltalk program.

All constants defined in Section 4.5.4 have initially the value undef, except Last_value which is equal to nil.

4.5.6 Recapitulation of the definition

We have already defined the entire algebra $\mathcal{A}_0$. Its superuniverse is equal to the union of universes:

$$\begin{aligned}&\textsc{Names} \cup \textsc{Lists\_of\_names} \cup \textsc{Objects} \cup \\ &\qquad \textsc{Variables} \cup \textsc{Methods} \cup \textsc{Expressions} \cup \textsc{States}.\end{aligned}$$

Chapter 5

Transition rule

In Chapter 4 we defined the algebra $\mathcal{A}_0$, which represents a 'static' part of our semantics of Smalltalk. In this chapter we give the 'dynamic' part of the semantics by defining a transition rule $\mathcal{R}_0$ which transforms the algebra $\mathcal{A}$ into another algebra $\mathcal{B}$. Both $\mathcal{A}$ and $\mathcal{B}$ have the same signature but different interpretation of some terms. If we apply $\mathcal{R}_0$ many times, then we obtain a series of algebras

$$\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i, \ldots$$

where $\mathcal{A}_{i+1}$ is obtained by applying $\mathcal{R}_0$ to $\mathcal{A}_i$.

The rule $\mathcal{R}_0$ is complex. Therefore we divide $\mathcal{R}_0$ into several small sub–rules, and give each of them a name. $\mathcal{R}_0$ is presented in the top–down style: from the general description of computation to technical details. The rule which is the 'top' of all other rules is called MAIN_RULE.

5.1 The MAIN_RULE

The rule MAIN_RULE is a main rule, i.e. each run starts with an analysis of conditions given in this rule. This rule divides $\mathcal{R}_0$ into three fragments, where each fragment represents different step of computation. These fragments will be described in the next sections. Now, we present only a brief explanation of their meaning, needed for understanding of the entire rule. The same solution will be used repeatedly, because many rules have the similar form.

While $\mathsf{Method} \neq \mathsf{undef}$, the rule STATE_CREATION is performed. Suppose that $\mathsf{Method} = \mathcal{M}$. In this rule, for a method represented by $\mathcal{M}$ new element $S \in \textsc{States}$ is created. In S all information associated with the execution of $\mathcal{M}$ is stored. The rule STATE_CREATION assigns S to the constant Current_state (so the condition $\mathsf{Current\_state} \neq \mathsf{undef}$ becomes satisfied) and defines, besides other things, a value of the term *Current_expression* (Current_state). After setting values of all functions defined on domains including $\textsc{States}$ on the argument S, the rule STATE_CREATION assigns undef to Method and Receiver. It is easy to notice that, from this point on, rules MAIN_RULE and STATE_CREATION are not equivalent any more.

The value of the term *Current_expression* (Current_state) plays important role in $\mathcal{R}_0$. We can think that in every moment some expression is evaluated. Evaluation of this expression can be in

---

Rule MAIN_RULE
 if Method $\neq$ undef **then**
  STATE_CREATION
 elseif Current_state = undef **then**
 elseif *Current_expression* (Current_state) $\neq$ undef **then**
  STATEMENT_EXECUTION
 else
  COMPLETION
 endif
EndRule

---

one of three stages: expression is prepared for computation of its value, expression is computed or value of this expression is passed to another expression. *Current_expression* (Current_state) is denoted shortly by $\underline{CurrExp}$, so

$$\underline{CurrExp} \stackrel{\text{df}}{=} Current\_expression\,(\text{Current\_state})$$

If $\underline{CurrExp} \neq$ undef and simultaneously Method = undef, then MAIN_RULE is equivalent to the rule STATEMENT_EXECUTION. Transitions of $\mathcal{A}$ explain how the method $\mathcal{M}$ and the statements of $\mathcal{M}$ are executed. STATEMENT_EXECUTION is finished by the rule $\underline{CurrExp}$:= undef included in this rule. At this moment values of constants Last_value and Return_prefix have enough informations about computations being executed, so execution of the last part of MAIN_RULE begins, and actions described in COMPLETION are performed.

It is easy to notice that execution of an expression of $\mathcal{M}$ may require execution of another method $\mathcal{M}$'. In such a situation the rule STATEMENT_EXECUTION assigns appropriate values to the constants Method and Receiver. It causes suspension of computation in the state S and creation of a new state S' which makes execution of $\mathcal{M}$' possible. After execution of $\mathcal{M}$', computations in S are continued and may use the result of execution of $\mathcal{M}$'. It implies that during transformations of $\mathcal{A}$ according to the program $\mathcal{R}_0$ a stack-like sequence of states is created. This sequence consists of elements

Current_state,
Previous_state (Current_state),
Previous_state (*Previous_state* (Current_state)),
...

The rule COMPLETION describes computations performed after execution of $\mathcal{M}$ (and $\mathcal{M}$'). This rule determines the final result of computation (i.e. determines the value of an expression containing a call of the method $\mathcal{M}$) using values of Last_value and Return_prefix, makes subsequent use of this result possible, and finishes computations in a state S determining a new value of Current_state. This is often a state in which a statement with the call of the method $\mathcal{M}$ is executed. It may also happen that the rule COMPLETION causes assigning undef to the constant Current_state. It is easy to notice that if Current_state is equal to undef, then Method = undef, unless the computation has been only just started. In such a case the subsequent transition of the algebra $\mathcal{A}$ is described by the empty rule, i.e. it does not change $\mathcal{A}$. In such a case the evolution of $\mathcal{A}$ terminates.

5.2 Creation and initialization of new states

The first step after beginning of execution of a method $\mathcal{M}$ is creation of a state in which $\mathcal{M}$ will be executed. In our semantics this is described by the rule STATE_CREATION. STATE_CREATION begins by selecting the kind of a method which will be performed.

Rule STATE_CREATION
  if Method $\in$ BLOCK_METHODS **then**
    if (*Argument_names* (Method), *Argument_names* (Receiver)) $\in$ EQUAL_LENGTHS **then**
      BLOCK_STATE_CREATION
    else
      ERROR_1
    endif
  else
    METHOD_STATE_CREATION
  endif
EndRule

Our semantics gives the meaning for two kinds of methods. These are ordinary methods and methods which execute blocks and which are represented by elements of the universe BLOCK_METHODS. In order to perform any method an element of STATES is created. In each case states have an identical structure, but they are initialized in different way. States created for execution of blocks are called block states. The others are called method states.

Before executing a block, arguments of the method which executes this block become objects referred to by the names of block arguments. The number of names of block arguments should be equal to the number of arguments. Therefore before creating a block state the rule STATE_CREATION tests whether the numbers of arguments of the block to be performed and of the method which performs this block are equal. If not, then the rule ERROR_1 is fired. It stops computation and gives an information about the kind of error. $\mathcal{R}_0$ has more rules like this. All of them are called ERROR with a suitable subscript and are not defined in our semantics. These rules correspond to procedures of the interpreter that produce run-time error messages.

5.2.1 Creation and initialization of method states

According to the rule METHOD_STATE_CREATION, creation and initialization of a method state consists of four steps. At the beginning a new state is created and values of some

Rule METHOD_STATE_CREATION
if Argument_names = undef **then**
  NEW_METHOD_STATE
  Argument_names := *Argument_names* (Method)
  Temporary_names := *Temporary_names* (Method)
  Argument := *First_argument* (<u>*CurrExp*</u>)
elseif Temporary_names ≠ empty **then**
  METHOD_VARIABLE_INITIALIZATION
  Temporary_names := *But_first* (Temporary_names)
elseif Argument_names ≠ empty **then**
  METHOD_ARGUMENT_PASSING
  Argument_names := *But_first* (Argument_names)
  Argument := *Next_argument* (Argument)
else
  Argument_names := undef
  Method := undef
  Receiver := undef
endif
EndRule

functions are defined. It is done by the rule NEW_METHOD_STATE. Additionally, names of both temporary variables and arguments, and the first argument of the method to be performed are stored in appropriate constants. Next, variables with temporary names and argument names are created, and initial values are assigned to them. This is done by the rules METHOD_VARIABLE_INITIALIZATION and METHOD_ARGUMENT_PASSING. Finally, rules Method := undef and Receiver := undef finish initialization of the state and then the computation of the method $\mathcal{M}$ begins.

The rule NEW_METHOD_STATE creates a new state. After creation, this state is pushed on the stack of all states and becomes the value of Current_state. The state which has been on the top of the stack is now a value of *Previous_state* (Current_state). As values of appropriate functions on the created state this rule also stores all informations necessary to perform the method. In this way this object keeps informations about receiver of the method $\mathcal{M}$ (the value of Receiver), the class of $\mathcal{M}$ represented by *Method_class* (Method) and the body of $\mathcal{M}$ represented by the value of *First_statement* (Method).

If $S \in$ STATES is a new method state, then *Home_state* (S) is equal to S. This property allows to distinguish both kinds of states. Values of the function *Home_state* : STATES → STATES are not changed during later transitions of the algebra $\mathcal{A}$. So it is obvious that if the value of Current_state is a method state, then the condition

$$\textit{Home\_state}\,(\text{Current\_state}) = \text{Current\_state}$$

holds. This condition will be denoted by the word <u>*is method state*</u>. We need the term

---

Rule NEW_METHOD_STATE
  extend STATES **with** new_state
    Previous_state (new_state) := Current_state
    Current_state := new_state
    Method_class (new_state) := *Method_class* (Method)
    Receiver (new_state) := Receiver
    Current_expression (new_state) := *First_statement* (Method)
    Home_state (new_state) := new_state
  endextend
EndRule

Home_state (Current_state) frequently, hence we denote it by *HomeState*, and we assume the following two definitions:

$$\underline{HomeState} \stackrel{\mathrm{df}}{=} Home\_state\,(\text{Current\_state})$$

$$\underline{is\ method\ state} \stackrel{\mathrm{df}}{=} \underline{HomeState} = \text{Current\_state}$$

Local variables are created before performing the method. In algebra $\mathcal{A}$ this is done by the rules METHOD_VARIABLE_INITIALIZATION and METHOD_ARGUMENT_PASSING. Both

---

Rule METHOD_VARIABLE_INITIALIZATION
  extend VARIABLES **with** new_variable
    Variable_value (new_variable) := nil
    Temporary_variable (*First* (Temporary_names), Current_state) := new_variable
  endextend
EndRule

Rule METHOD_ARGUMENT_PASSING
  extend VARIABLES **with** new_variable
    Variable_value (new_variable) :=
        Expression_value (Argument, *Previous_state* (Current_state))
    Temporary_variable (*First* (Argument_names), Current_state) := new_variable
  endextend
EndRule

---

rules create necessary variables, assign them to their names (i.e. define values of the function *Temporary_variable* $(\cdot,$Current_state$)$), and assign initial values. In Smalltalk, temporary variables are initialized as nil. Values of variables with argument names are initialized using results computed in the state *Previous_state* (Current_state). We may say that they are transferred from this state to Current_state.

The rule METHOD_STATE_CREATION initializes all the terms which are necessary in computation of the method. In Smalltalk, it is possible to declare in the same method two variables with the same name w: as a temporary variable and as a block argument. In such a case rules METHOD_VARIABLE_INITIALIZATION and METHOD_ARGUMENT_PASSING create a variable with the name w, and fix its initial value. The final value of *Temporary_variable* on w and Current_state is fixed by the second rule, i.e. METHOD_ARGUMENT_PASSING. Hence, w refers to the argument described in the message invoking $\mathcal{M}$. More complicated case occurs when w is, for instance, also a name of an instance variable. In this case correct meaning of the variable with the name w is determined by a rule PRIVATE_VARIABLE.

5.3 Creation and initialization of block states

Let us consider a block

$$[\ :x_1\ \ldots\ :x_n\ \mid\ s_1 .\ \ldots\ .s_m]$$

which is a part of a method M. $x_1, \ldots, x_n$ are names of block arguments, and $s_1, \ldots, s_m$ are statements. On one hand, block argument names are names of arguments of block, and play similar role as argument names in methods. On the other hand, names of block arguments are treated as temporary names of method M. Even if these names are not declared, they may be used in every place of M. In our semantics of Smalltalk, this fact is expressed by the definition of the function *Temporary_names* given in Section 4.2.3.3.

Statements included in block expression in the method M can use the same variables as other statements in M, e.g. temporary variables, object variables, class variables or pool variables. It means, that if M has been performed in a state S for some object O, and a block B has been created in S, then statements of B can use temporary variables created in S, object variables of O and class and pool variables of class of O, although B is performed in a state different from S. To make it possible, a block holds the information where it has been created. If a block B is created during execution of the method M to which it belong, i.e. in a state S, then the state S is the value of the term *Home_state* (B) (see Section 5.5.2). It allows B to refer to variables available in S, i.e. the state *Home_state* (B) enables us to find variables needed during execution of B.

Creation and initialization of a block state is done by the rule BLOCK_STATE_CREATION. First this rule creates a new state using the rule NEW_BLOCK_STATE. Apart from state creation, this rule, like NEW_METHOD_STATE, puts the new state on the top of the stack of states and defines for it values of the functions

$$Current\_expression : \textsc{States} \rightarrow \textsc{Expressions}$$

and

$$Home\_state : \textsc{States} \rightarrow \textsc{States}.$$

The first of these functions determines all statements of the block to be executed. The second assigns the state in which the block was created to the new state. By this function, a block state

Rule BLOCK_STATE_CREATION
 if Argument_names = undef **then**
  NEW_BLOCK_STATE
  Argument_names := *Argument_names* (Receiver)
  Argument := *First_argument* (*CurrExp*)
 elseif Argument_names $\neq$ empty **then**
  BLOCK_ARGUMENT_PASSING
  Argument_names := *But_first* (Argument_names)
  Argument := *Next_argument* (Argument)
 else
  Argument_names := undef
  Method := undef
  Receiver := undef
  Last_value := nil
 endif
EndRule

Rule NEW_BLOCK_STATE
 extend STATES **with** new_state
  Home_state (new_state) := *Home_state* (Receiver)
  Current_expression (new_state) := *First_statement* (Receiver)
  Previous_state (new_state) := Current_state
  Current_state := new_state
 endextend
EndRule

has access to variables used when the executed block was created. Notice that the condition *is method state* (see Section 5.2.1) does not hold for block states.

During initialization of a block state S, new variables are not created. Instead, arguments of the block are initialized using results obtained in the state *Previous_state* (S). We remember here, that arguments of block are temporary variables of a state *Home_state* (S). New values of these variables are initialized by the rule BLOCK_ARGUMENT_PASSING.

Initialization of a block state is finished when the assignment Method := undef from the rule BLOCK_STATE_CREATION is done. Then the rule $\mathcal{R}_0$ starts to execute statements of the block.

Rule BLOCK_ARGUMENT_PASSING
Variable_value (*Temporary_variable* (*First* (Argument_names), $\underline{HomeState}$)) :=
Expression_value (Argument, *Previous_state* (Current_state))
EndRule

5.4 Usage of variables

Before providing the next sub–rule contained in MAIN_RULE, we give two rules which explain how variables are used in Smalltalk. The rule VARIABLE_DEREFERENCE finds the value of a variable with the name *Name* ($\overline{CurrExp}$) and stores it as a value of the term $\overline{CurrExpVal}$. The definition of $\overline{CurrExpVal}$ can be found in Section 5.5. The rule ASSIGNMENT assigns the value of $\overline{CurrExpVal}$ to the variable with the name *First* (Assignments). Now, we give the rule VARIABLE which is used in both rules mentioned above.

5.4.1 Interpretation of variable names

The rule VARIABLE finds a variable of the name Variable_name and stores it in the constant Variable. This rule appears only in VARIABLE_DEREFERENCE and ASSIGNMENT, and is preceded by rules fixing values of the constants Pool_names and Class (see sections 5.4.4 and 5.4.5). At the very beginning the constant Class stores either the class of the currently performed method or the class of the method which contains the definition of the currently executed block. The constant Pool_names stores a list of names of dictionaries available in the class Class. In all cases, informations stored in Variable_name, Class and Pool_names determine certain variables uniquely. Class and Pool_names are used only when names of shared variables are interpreted.

In Smalltalk names of shared variables are capitalized, while the names of private variables are not. This principle allows us to consider different kinds of variables separately, and guarantee correctness of the rule VARIABLE.

Rule VARIABLE
if Variable_name ∈ SHARED_VARIABLE_NAMES **then**
SHARED_VARIABLE
else
PRIVATE_VARIABLE
endif
EndRule

5.4.2 Interpretation of shared variable names

Interpretation of shared variable names is done by the rule SHARED_VARIABLE. In Smalltalk,

---

Rule SHARED_VARIABLE
  if *Global_variable* (Variable_name) $\neq$ undef **then**
    Variable := *Global_variable* (Variable_name)
  elseif Pool_names $\neq$ empty **then**
    if *Pool_variable* (Variable_name, *First* (Pool_names)) $\neq$ undef **then**
      Variable := *Pool_variable* (Variable_name, *First* (Pool_names))
    else
      Pool_names := *But_first* (Pool_names)
    endif
  elseif *Class_variable* (Variable_name, Class) $\neq$ undef **then**
    Variable := *Class_variable* (Variable_name, Class)
    Class := undef
  elseif *Superclass* (Class) $\neq$ undef **then**
    Class := *Superclass* (Class)
  else
    ERROR_2
  endif
EndRule

---

there are three kinds of shared variables: global, dictionary and class variables. For each kind, there is a function which assigns an appropriate variable to a given name, and which is defined for some name if and only if a variable with this name exists. The rule SHARED_VARIABLE checks in turn, whether the function *Global_variable* is defined for Variable_name, whether the function *Pool_variable* is defined for the argument containing Variable_name and one of dictionaries mentioned on the list Pool_names. Finally SHARED_VARIABLE checks whether the function *Class_variable* is defined for the argument containing Variable_name and either the class Class or one of its superclasses. The name Variable_name is interpreted as a name of the first variable found in this way.

It is easy to notice that SHARED_VARIABLE defines a way of interpretation with a name W in the case when many shared variables of the name W coexist. If a global variable of the name W exists, then W is interpreted always as a name of this global variable. If W is simultaneously a name of a pool variable and a class variable, then it is interpreted as a name of a pool variable. This rule takes into account inheritance of class variables and hiding both pool and class variables. During performance of the method $\mathcal{M}$, it is possible to use variables of the class of $\mathcal{M}$ and variables inherited from all superclasses of the class of $\mathcal{M}$.

If there is no shared variable with a given name, then SHARED_VARIABLE fires the rule ERROR_2. Its execution should abandon computation and give an information that an appropriate variable was not found. We do not give this rule here.

5.4.3 Interpretation of private variable names

In the algebra $\mathcal{A}_0$ there are three functions which map names into private variables. Local variables and variables corresponding to both method and block arguments are values of the function *Temporary_variable*. Pseudovariables different from *self* and *super* are values of *Pseudovariable*. The function *Object_variable* assumes object variables as values. The rule PRIVATE_VARIABLE, like SHARED_VARIABLE, checks whether these functions are defined

---

Rule PRIVATE_VARIABLE
  if *Pseudovariable* (Variable_name) $\neq$ undef **then**
    Variable := *Pseudovariable* (Variable_name)
 elseif *Object_variable* (Variable_name, *Receiver* (<u>*HomeState*</u>)) $\neq$ undef **then**
    Variable := *Object_variable* (Variable_name, *Receiver* (<u>*HomeState*</u>))
 elseif *Temporary_variable* (Variable_name, <u>*HomeState*</u>) $\neq$ undef **then**
    Variable := *Temporary_variable* (Variable_name, <u>*HomeState*</u>)
 else
    ERROR_3
 endif
EndRule

---

for Variable_name. If one of them is defined for this name, then the first value that was found is stored in Variable. If there is no private variable with a given name, then PRIVATE_VARIABLE fires the rule ERROR_3.

This rule implies, that in Smalltalk a local variable is hidden by an object variable with the same name. Moreover, both an object variable and a local variable are hidden by a pseudovariable with the same name. In the case of private variables, the error of a missing variable should not occur. In real compiler, a semantics analysis is done for every new method, and all occurrences of private variables are checked up.

5.4.4 Variable dereference

The rule VARIABLE_DEREFERENCE finds a variable with the name *Name* (<u>*CurrExp*</u>) and stores its value into the term <u>*CurrExpVal*</u>. This rule works in two different ways depending on the input.

Among Smalltalk variables, there are special ones named *self* and *super*. Values of these variables are not stored and are equal to *Receiver* (<u>*HomeState*</u>). The variable *self* allows sending a message to a receiver to an object which is called to perform the method. More precisely, it means that if a computation is in a method state, then *self* refers to the receiver of the currently performed message. For a block state *S*, the value of this variable is equal to the receiver of the message that caused creation of the block executed in the state *S*. A value of *super* is

Rule VARIABLE_DEREFERENCE
 if *Name* (<u>*CurrExp*</u>) = self **then**
  <u>*CurrExpVal*</u> := *Receiver* (<u>*HomeState*</u>)
  Assignments := *Assignments* (<u>*CurrExp*</u>)
 elseif *Name* (<u>*CurrExp*</u>) = super **then**
  <u>*CurrExpVal*</u> := *Receiver* (<u>*HomeState*</u>)
  Assignments := *Assignments* (<u>*CurrExp*</u>)
 elseif Variable_name = undef **then**
  Variable_name := *Name* (<u>*CurrExp*</u>)
  Pool_names := *Shared_pools* (*Method_class* (<u>*HomeState*</u>))
  Class := *Method_class* (<u>*HomeState*</u>)
 elseif Variable = undef **then**
  VARIABLE
 else
  <u>*CurrExpVal*</u> := *Variable_value* (Variable)
  Assignments := *Assignments* (<u>*CurrExp*</u>)
  Variable := undef
  Variable_name := undef
 endif
EndRule

defined in the same way. Usage of *super* causes some additional effects which are presented in Section 5.5.3.

Other names of variables are first interpreted by the rule VARIABLE. If VARIABLE has been finished, the constant Variable refers to an element of VARIABLES which represents an appropriate variable. Then the object *Variable_value* (Variable) is assigned to the term <u>*CurrExpVal*</u>. Comments concerning the constant Assignments are given in Section 5.4.5.

5.4.5 Assigning a value to a variable

Assignment of the value of the term <u>*CurrExpVal*</u> to the variable of the name *First* (Assignments) is done by the rule ASSIGNMENT. The variable found by the rule VARIABLE for the name *First* (Assignments) is stored in the constant Variable. The rule ASSIGNMENT changes the value of the term *Variable_value* (Variable) which represents a value of Variable, and fires the rule

Assignments := *But_first* (Assignments)

which allows an execution of the next assignment or provides information that all assignments are done.

Rule ASSIGNMENT
  if Variable_name = undef **then**
    Variable_name := *First* (Assignments)
    Pool_names := *Shared_pools* (*Method_class* (*HomeState*))
    Class := *Method_class* (*HomeState*)
  elseif Variable = undef **then**
    VARIABLE
  else
    Variable_value (Variable) := *CurrExpVal*
    Assignments := *But_first* (Assignments)
    Variable := undef
    Variable_name := undef
  endif
EndRule

5.5 Evaluating expressions

In the most imperative languages, the statements and the expressions are different language constructions. If an expression is evaluated, then it returns its value, but if a statement is computed, then no value of this statement is expected. In contrast to these languages, in Smalltalk the notion of expression and the notion of statement are very similar, because each statement has a value. Therefore each statement can be used in any context where an expression is expected.

Let us illustrate this by an example. Consider two statements[1]

```
x ← x + 1
```

and

```
y ← 2 * y
```

which assign the values of `'x + 1'` and `'2 * y'` to the variables `x` and `y`, repectively. Both statements can be used as expressions, e.g. in the following message expression

```
(x ← x + 1) gcd:  (y ← 2 * y)
```

where the statement `'x ← x + 1'` denotes here the receiver of the message `gcd:`, and the statement `'y ← 2 * y'` denotes an argument of this message expression. As the value of assignment, the value of the right–hand side expression is used. Therefore this message expression is similar to the expression

```
(x + 1) gcd:  (2 * y)
```

but some differences may occur because of side–effects caused by assignments.

[1]Statements are words derived from ⟨*Statement*⟩, see Section 4.3.2

These remarks above suggest, that in Smalltalk statements and expressions can be treated in the uniform way. Therefore in the rule $\mathcal{R}_0$ statements are not distinguished from expressions.

We often need a term *Expression_value* ($\underline{CurrExp}$, Current_state), so we use the following abbreviation

$$\underline{CurrExpVal} \stackrel{\text{df}}{=} Expression\_value\,(\underline{CurrExp}, \mathsf{Current\_state}).$$

When evaluation of $\underline{CurrExp}$ is finished, the value of $\underline{CurrExp}$ is stored as a value of *Expression_value* for appropriate arguments by firing the rule $\underline{CurrExpVal} := \ldots$

The rule STATEMENT_EXECUTION computes the value of the expression E represented by $\underline{CurrExp}$ and causes this value to be referred to by all names from the list *Assignments* ($\underline{CurrExp}$). In fact during execution of STATEMENT_EXECUTION, the term $\underline{CurrExp}$ assumes also other values from the set EXPRESSIONS, other expressions are evaluated.

Rule STATEMENT_EXECUTION
  if $\underline{CurrExpVal}$ = undef **then**
    EXPRESSION_EVALUATION
  elseif Assignments ≠ empty **then**
    ASSIGNMENT
  elseif *Superexpression* ($\underline{CurrExp}$) ≠ undef **then**
    $\underline{CurrExp}$:= *Superexpression* ($\underline{CurrExp}$)
  elseif *Next_argument* ($\underline{CurrExp}$) ≠ undef **then**
    $\underline{CurrExp}$:= *Next_argument* ($\underline{CurrExp}$)
  elseif *Next_statement* ($\underline{CurrExp}$) ≠ undef **then**
    $\underline{CurrExp}$:= *Next_statement* ($\underline{CurrExp}$)
  else
    Last_value := $\underline{CurrExpVal}$
    Return_prefix := *Return_prefix* ($\underline{CurrExp}$)
    $\underline{CurrExp}$:= undef
  endif
EndRule

In the first step the rule STATEMENT_EXECUTION checks whether the value of $\underline{CurrExpVal}$ is defined. If not, then the value of an expression represented by $\underline{CurrExp}$ is not computed yet, and further computations consist of execution of the rule EXPRESSION_EVALUATION. This rule is written in such a way that setting a value of $\underline{CurrExpVal}$ is done simultaneously with assigning the value of the term *Assignments* ($\underline{CurrExp}$) to the constant Assignments. In the next step, if $\underline{CurrExpVal}$ is updated, appropriate assignments are done by the rule ASSIGNMENT defined in Section 5.4.5.

If all assignments in ASSIGNMENT are done, then STATEMENT_EXECUTION tests if one of the functions *Superexpression* , *Next_argument* or *Next_statement* is defined on $\underline{CurrExp}$. If it is true, then not all expressions of the method being performed are evaluated yet, so an appropriate update rule causes change of $\underline{CurrExp}$ and evaluation of the next expression. Otherwise, all expressions are evaluated, and $\underline{CurrExp}$ stores the value of last expression E of this method. In this case, sufficient informations about results of computation are stored in the constants Last_value and Return_prefix. The value of E is referred to by Last_value. Return_prefix stores information whether E is a part of the return statement contained in the body of the method to be performed. The values of these constants allow us to determine which values are returned and from which state the computation should be continued. Setting the value of $\underline{CurrExp}$ equal to undef causes that further computations are continued in according to the rule COMPLETION.

5.5.1 Expression type recognition

The evaluation of expressions depends on their type. We classify expressions as a primary or a message (including cascade message) expression. By the definition of the function *Receiver* , this function is defined for elements corresponding to message expressions, and only for these elements. In order to verify whether $\underline{CurrExp}$ does not correspond to a message expression, we test a condition $\underline{is\ not\ method}$ which is defined as follows:

$$\underline{is\ not\ method} \stackrel{\text{df}}{=} Receiver\,(\underline{CurrExp}) = \mathsf{undef}.$$

Now, we consider primary expressions. We have three kinds of primary expressions: literal constants, variable names and block definitions. If $\mathsf{e} \in$ EXPRESSIONS corresponds to a literal constant, then the function *Constant_value* is defined on e. Therefore we define a condition:

$$\underline{is\ literal} \stackrel{\text{df}}{=} Constant\_value\,(\underline{CurrExp}) \neq \mathsf{undef}.$$

Elements of EXPRESSIONS which correspond to primary expressions, for which the function *Name* is defined, correspond to variable names. We define

$$\underline{is\ variable} \stackrel{\text{df}}{=} Name\,(\underline{CurrExp}) \neq \mathsf{undef}.$$

So, conditions $\underline{is\ variable}$ and $\underline{is\ not\ method}$ decide whether $\underline{CurrExp}$ corresponds to a variable name. If $\mathsf{e} \in$ EXPRESSIONS corresponds to a primary expression and neither to a literal nor to a variable, then e corresponds to a block definition.

Two rules which are presented in this section test the type of $\underline{CurrExp}$ and compute $Expression\_value\,(\underline{CurrExp},\mathsf{Current\_state})$ in an appropriate way. The value of the term $Expression\_value\,(\underline{CurrExp},\mathsf{Current\_state})$ represents the value of the expression represented by $\underline{CurrExp}$ computed in the current state. In the first step the rule EXPRESSION_EVALUATION tests the condition $\underline{is\ not\ method}$. If it is true, then the rule PRIMARY_EVALUATION is evaluated.

If this condition does not hold, then $\underline{CurrExp}$ corresponds to a message expression. In such a case in the next step EXPRESSION_EVALUATION verifies if the value of $Receiver\,(\underline{CurrExp})$ is computed. If not, then it changes $\underline{CurrExp}$, and causes computation of its value.

In the next step EXPRESSION_EVALUATION tests whether $First\_argument\,(\underline{CurrExp})$ is defined. If it is defined, then $\underline{CurrExp}$ represents a non-unary message expression,

---

Rule EXPRESSION_EVALUATION
if $\underline{\textit{is not method}}$ **then**
PRIMARY_EVALUATION
elseif $\textit{Expression\_value}\,(\textit{Receiver}\,(\underline{\textit{CurrExp}}), \text{Current\_state}) = \text{undef}$ **then**
$\underline{\textit{CurrExp}} := \textit{Receiver}\,(\underline{\textit{CurrExp}})$
elseif $\textit{First\_argument}\,(\underline{\textit{CurrExp}}) = \text{undef}$ **then**
METHOD_LOOKUP
elseif $\textit{Expression\_value}\,(\textit{First\_argument}\,(\underline{\textit{CurrExp}}), \text{Current\_state}) = \text{undef}$ **then**
$\underline{\textit{CurrExp}} := \textit{First\_argument}\,(\underline{\textit{CurrExp}})$
else
METHOD_LOOKUP
endif
EndRule

---

and further computations of a value of $\underline{\textit{CurrExp}}$ require verifying whether the value of $\textit{First\_argument}\,(\underline{\textit{CurrExp}})$ is already computed. If it is not computed, then EXPRESSION_EVALUATION forces computation of $\textit{First\_argument}\,(\underline{\textit{CurrExp}})$ by changing $\underline{\textit{CurrExp}}$. The set EXPRESSIONS and the rule STATEMENT_EXECUTION are defined in such a way, that the other arguments are evaluated just after evaluation of the first one. So, if EXPRESSION_EVALUATION states that the value of $\textit{First\_argument}\,(\underline{\textit{CurrExp}})$ is defined, then values of other arguments are also defined.

The rule PRIMARY_EVALUATION examines the type of the expression $\underline{\textit{CurrExp}}$ under the

---

Rule PRIMARY_EVALUATION
if $\underline{\textit{is literal}}$ **then**
$\underline{\textit{CurrExpVal}} := \textit{Constant\_value}\,(\underline{\textit{CurrExp}})$
$\text{Assignments} := \textit{Assignments}\,(\,\underline{\textit{CurrExp}})$
elseif $\underline{\textit{is variable}}$ **then**
VARIABLE_DEREFERENCE
else
BLOCK_CREATION
endif
EndRule

---

assumption that this element represents a primary expression, and finds the value of $\underline{\textit{CurrExp}}$. Values of elements representing literal constants are set using the function *Constant_value*. Simultaneously the value of Assignments is changed in according to the principle given by the rule STATEMENT_EXECUTION (see Section 5.5). Values of elements representing variable names are determined by VARIABLE_DEREFERENCE given in Section 5.4.4. In other cases, $\underline{CurrExp}$ represents a block definition, and its value is computed by the rule BLOCK_CREATION.

5.5.2 Evaluation of block definitions

Values of block definitions are objects of a special class. They are created during execution of either methods or blocks containing these definitions. Values of such expressions are objects of the class Block, and are created by the rule BLOCK_CREATION. This rule sets values of all func-

---

Rule BLOCK_CREATION
  extend OBJECTS **with** new_block
    $\underline{CurrExpVal}$:= new_block
    Class (new_block) := block
    Home_state (new_block) := $\underline{HomeState}$
    First_statement (new_block) := *First_statement* ($\underline{CurrExp}$)
    Argument_names (new_block) := *Argument_names* ($\underline{CurrExp}$)
  endextend
  Assignments := *Assignments* ($\underline{CurrExp}$)
EndRule

---

tions defined on BLOCKS, and describes elements of this set. Functions *Argument_names* and *First_statement* stores some informations provided by $\underline{CurrExp}$, e.g. the list of block arguments and the element of EXPRESSIONS corresponding to the first statement of the block.

If M is a method performed in a state S, and M includes a block definition, and block B is created in S, then

$$Home\_state\,(B) = S$$

BLOCK_CREATION sets also the constant Assignments according to the principle given in Section 5.5.

5.5.3 Message expressions

Suppose that $\underline{CurrExp}$ is an element of EXPRESSIONS representing a message expression E. For such an element, functions *Receiver* and *Name* are defined. The expression E is interpreted as a request for performing a method M. The method M is one of the method with the name *Name* ($\underline{CurrExp}$) and which belong to the set of methods depending on *Receiver* ($\underline{CurrExp}$). Computation of the value of E requires execution of M. If the method M is known, then it is enough to assign simultaneously M as a value of Method and the value of *Receiver* ($\underline{CurrExp}$) to Receiver. After firing the appropriate update rules, the rule $\mathcal{R}_0$ creates a new state S and performs all statements of M. Next the rule COMPLETION starts. This rule determines the return value for the current activation of the method M, removes the state S, and assigns the

Rule METHOD_LOOKUP
  if Class = undef **then**
    INITIAL_CLASS
  elseif *Method* (*Name* ($\underline{CurrExp}$), Class) = undef **then**
    if *Superclass* (Class) $\neq$ undef **then**
      Class := *Superclass* (Class)
    else
      ERROR_4
    endif
  else
    Method := *Method* (*Name* ($\underline{CurrExp}$), Class)
    Receiver := *Expression_value* (*Receiver* ($\underline{CurrExp}$), Current_state)
    Class := undef
  endif
EndRule

return value to $\underline{CurrExpVal}$. Now, it remains to explain how Smalltalk finds the method to be performed. This is done by the rule METHOD_LOOKUP.

In order to find a method M with the given signature, Smalltalk searches the class hierarchy from some class upwards. The method M is defined as the first method of the given name found in this way. The rule METHOD_LOOKUP first fixes the class which is the starting point of searching, and assigns it to Class. This is realized by the rule INITIAL_CLASS.

In the next step METHOD_LOOKUP verifies in turn if a method of the name *Name* ($\underline{CurrExp}$) exists in classes Class, *Superclass* (Class), *Superclass* (*Superclass* (Class)), and so on. If such a method does not exist, then the error representing a call of unknown method is reported by the rule ERROR_4. As in the previous case, we do not define ERROR_4. If such a method exists, then METHOD_LOOKUP stores it in the constant Method, and simultaneously stores the value of *Receiver* ($\underline{CurrExp}$) in Receiver, and the rule $\mathcal{R}_0$ starts perform the method stored in Method.

According to the rule INITIAL_CLASS, in most cases, the method to be performed is searched from the class of the receiver of the evaluated expression, i.e. from the class *Class* (*Expression_value* (*Receiver* ($\underline{CurrExp}$),Current_state)). If expression *super* is the receiver in the evaluated expression E, then the look up method starts from the superclass of the class of the method which contains E. Precisely speaking, look up starts from *Superclass* (*Method_class* ($\underline{HomeState}$)). If the computation is in a block state, then look up starts from the superclass of the class of the method containing the definition of the block being executed. It may happen that the superclass of *Method_class* ($\underline{HomeState}$) does not exist. Then the rule INITIAL_CLASS reports the appropriate error.

Rule INITIAL_CLASS
if *Name* (*Receiver* (<u>*CurrExp*</u>)) $\neq$ super **then**
Class := *Class* (*Expression_value* (*Receiver* (<u>*CurrExp*</u>), Current_state))
elseif *Superclass* (*Method_class* (<u>*HomeState*</u>)) = undef **then**
ERROR_5
else
Class := *Superclass* (*Method_class* (<u>*HomeState*</u>))
endif
EndRule

5.6 Termination of computation

After execution of all statements of either a method or a block, the value of the last evaluated statement and information whether this statement is a return expression (such an expression starts with an up arrow $\uparrow$) are stored by the rule STATEMENT_EXECUTION in constants Last_value and Return_prefix, respectively. Now, using these constants, the value returned by a method or a block is determined, and passed to further computations. Moreover, after executing all statements some states from the stack will be removed. States to be removed will be called superfluous. These actions are realized by the rule COMPLETION.

Rule COMPLETION
if Superfluous_state = undef **then**
SETTING_SUPERFLUOUS_STATE
elseif <u>*ReturnState*</u> = undef **then**
Current_state := undef
elseif Current_state $\neq$ Superfluous_state **then**
STATE_DELETION
Current_state := *Previous_state* (Current_state)
else
VALUE_PASSING
Superfluous_state := undef
Return_prefix := undef
Last_value := nil
endif
EndRule

First, the rule COMPLETION performs the rule SETTING_SUPERFLUOUS_STATE which finds the last state which should be removed from stack, and stores it in Superfluous_state.

Now, the value of Superfluous_state allows to fix a state, which is on the top of the stack after the rule COMPLETION has been realized. This state is the value of the term

$$\underline{\mathit{ReturnState}} \stackrel{\text{df}}{=} \mathit{Previous\_state}\,(\mathsf{Superfluous\_state}).$$

It may happen that all states should be removed from the stack. In this case, ***ReturnState*** is undefined, and COMPLETION assigns undef to Current_state. At such a moment, the equality Method = undef holds, too. Hence, it breaks the transition of $\mathcal{A}$ described by the rule $\mathcal{R}_0$. Otherwise, COMPLETION deletes the appropriate fragment of the stack realizing the rule STATE_DELETION, and executes VALUE_PASSING which computes and transmits the returned value and changes Current_state. In this way the returned value becomes the value of an appropriate expression and computation is continued in another state.

5.6.1 Executing return statements and superfluous states

In order to perform a method, a certain state S is created and pushed onto the stack. After completion of this method, the state S becomes superfluous and should be removed. Current_state is the only superfluous state after finishing computations in a method state. In the case of finishing computation in a block state, there may be more than one superfluous state. The number of these states depends on whether a return statement has been executed.

A return statement may occur only at the end of either a method or a sequence of statements occurring in a block definition. Execution of a return statement occurring in the body of a method M (either at the end of the method or at the end of an block definition included in the method M), besides other things, always stops execution of M. Moreover, it is the unique way of abandoning of M before executing of all statements. Executing all statements of a block without a return statement finishes only some part of computations realized by M, and the remaining computations are continued. In this section, we present how these principles are realized by the rule $\mathcal{R}_0$. Other consequences of executing return statements are given in Section 5.6.3.

Now, consider the following three methods:

m1	**m2**	**m3**

`Var := [[↑5] value. ↑4].`
`O m2.`
`↑1`
	`O m3.`
	`↑2`
		`Var value.`
		`↑3`

Suppose that these methods are of the same class as the object `O`, and *Var* is a global variable. Assume also that `'value'` is a signature of a method which performs blocks (belongs to BLOCK_METHODS). We will examine how the expression `'O m1'` included in the body of a method performed in some state S_0 will be evaluated. Below we give a table which shows these computations.

Current state	method/block	current expression	created state
S_0		`0 m1`	S_1
S_1	`m1`	`0 m2`	S_2
S_2	`m2`	`0 m3`	S_3
S_3	`m3`	`Var value`	S_4
S_4	`[[↑5] value. ↑4]`	`[↑5] value`	S_5
S_5	`[↑5]`	`↑5`	

Evaluation of the expression `'0 m1'` causes creation of a new state S_1. In this state, the method `m1` is executed. In some moment during executing `m1`, the expression `'0 m2'` starts to be evaluated. It causes creation of a state S_2, and next, similarly, states S_3, S_4 and S_5. Notice that S_4 and S_5 are block states. In these states statements given in `m1` are executed, and among them the return statement `'↑5'`. Ending of computations in S_5 implies ending of computations in the state S_1. It also makes further computations in states S_2, S_3 and S_4 unnecessary. So, all states from S_1 to S_5 are superfluous. Notice that the states S_5 holds S_1 as *Home_state* (S_5). It is obvious that in the state S_1, the block B_1 including statements `'[↑5] value. ↑4'` is created. For this block *Home_state* (B_1) is defined as S_1. The block B_1 becomes the value of the variable `Var`. In the state S_4 statements of B_1 are executed. During computation of a receiver from the first statement, a block B_2 containing the statement `'↑5'` is created. We have

$$\mathit{Home\_state}\,(B_2) = \mathit{Home\_state}\,(S_4) = \mathit{Home\_state}\,(B_1) = S_1.$$

The block B_2 is executed in the state S_5, so the following equations hold

$$\mathit{Home\_state}\,(S_5) = \mathit{Home\_state}\,(B_2) = S_1.$$

It means that execution of the return statement in the state S_5 should cause deletion of the states up to and including *Home_state* (S_5).

Deletion of states may cause additional troubles. It is possible that the statement `'Var value'` is executed in the state S_0 after `'0 m1'` (or in any other state). Evaluation of this expression needs creation of a state S_6. It will be a block state which satisfies

$$\mathit{Home\_state}\,(S_6) = \mathit{Home\_state}\,(B_1) = S_1.$$

During the further computation the return statement `↑5` will be executed. It should abandon computations in the state *Home_state* $(S_6) = S_1$ again, and delete from the stack states between S_1 and S_6. We assume, that such situation causes a run-time error.

It is easy to notice that in more complex cases, computations in S_6 need informations stored in S_1, for instance when the method `m1` uses local variables occurring also in the block B_1. Such a computation should not end in an error, if the last executed statement is not a return one. It implies that after ending computations in some state, this state may be still needed, and should exist.

In order to make possible finding errors mentioned in the previous paragraph, we divide states into active and the others. A state S is active, if computation in S has not been finished and will be continued. Computations in superfluous states are stopped and are not continued. Hence, these states always become not active. We will verify whether a state S is active by examining the condition *Current_expression* (S) $\neq$ undef. States created by the rule STATE_CREATION are active except for these that were created for execution of the empty method or block, and stop to be active when they are removed from the stack.

All details concerning fixing superfluous states are determined by the rule SETTING_SUPERFLUOUS_STATE. This rule finds the last superfluous state and stores it in the

---

Rule SETTING_SUPERFLUOUS_STATE
if Return_prefix = undef **then**
Superfluous_state := Current_state
elseif <u>*is method state*</u> **then**
Superfluous_state := Current_state
elseif *Current_expression* (<u>*HomeState*</u>)) = undef **then**
ERROR_6
else
Superfluous_state := <u>*HomeState*</u>
endif
EndRule

---

constant Superfluous_state. If in some state (either a block or a method one) computations do not end in executing a return statement, then only current state is superfluous and is removed. The case of a method state is similar, even if the method executed in this state is finished by a return statement. After execution of a block with a return statement, all states up to <u>*HomeState*</u> are superfluous, but they can be removed only in the case when <u>*HomeState*</u> is active. Otherwise, transitions of $\mathcal{A}$ are terminated by the rule ERROR_6 raising the run-time error.

5.6.2 State deletion

The stack which is used can be found at any moment by computing the following sequence:

Current_state,
Previous_state (Current_state),
Previous_state (*Previous_state* (Current_state)),
...

In order to remove some elements from this sequence, it is sufficient to assign one of these states to Current_state. According to the assumptions given in Section 5.6.1, we should take care that states outside the stack are not active when are being removed. Notice that before removing, the current state is not active.

The rule STATE_DELETION makes the state *Previous_state* (Current_state) inactive. The rule COMPLETION replaces Current_state by *Previous_state* (Current_state). It ensures that computations will not start in an inappropriate state. These acts are performed repeatedly until Current_state becomes the last superfluous state. Removing superfluous states from the stack by a change of Current_state is realized by VALUE_PASSING.

Rule STATE_DELETION
Current_expression (*Previous_state* (Current_state)) := undef
EndRule

5.6.3 Passing a value of a message expression

Before termination of transitions in accordance with the rule COMPLETION, three update rules from VALUE_PASSING are fired. First of these rules ends computation of the value of a message

Rule VALUE_PASSING
if Return_prefix = undef **then**
  if <u>*is method state*</u> **then**
    Expression_value (*Current_expression* (<u>*ReturnState*</u>), <u>*ReturnState*</u>) :=
      Receiver (Current_state)
  else
    Expression_value (*Current_expression* (<u>*ReturnState*</u>), <u>*ReturnState*</u>) := Last_value
  endif
else
  Expression_value (*Current_expression* (<u>*ReturnState*</u>), <u>*ReturnState*</u>) := Last_value
endif
Assignments := *Assignments* (*Current_expression* (<u>*ReturnState*</u>))
Current_state := <u>*ReturnState*</u>
EndRule

expression which caused execution of the method just finished, and stores the value of this expression as a value of the function *Expression_value* for appropriate arguments. The second one assigns the appropriate list to the constant Assignments in accordance with assumptions given in Section 5.5. The last one fixes a new value of Current_state and removes superfluous states from the stack. The rule COMPLETION simultaneously assigns default values to the constants Superfluous_state and Last_value preparing them for use in further computations.

Now, we shall explain how values of message expressions are defined. Every state S except of the first one pushed on the stack is created to compute the value of the expression *Current_expression* (*Previous_state* (S)). Let S be the state in which computations have just ended, and let us assume that *Previous_state* (S) is defined. Then just after termination of computation in S, *Current_expression* (*Previous_state* (S)) differs from undef. Its value will be denoted by e.

Let us suppose that computations in the state S do not end with execution of a return statement, so after ending computation in S the condition Return_prefix = undef holds. Then the value of the expression e (in the state *Previous_state* (S)) depends on the type of a method executed in S and, consequently, on the type of S. If the state S is a method one, then the value of e is equal to *Receiver* (S). Otherwise, it is equal to the value of the last expression evaluated in the state S that is stored in Last_value. Let us also notice that in such a case *ReturnState* is equal to *Previous_state* (S), so the rule VALUE_PASSING defines the value of e.

Let us notice that if the empty block was executed in S, then Last_value was not changed by STATEMENT_EXECUTION (see MAIN_RULE). In this case, the value of e is equal to the initial value of Last_value, so it is nil.

Let us suppose now, that computation in S ends with execution of the return statement. Then the computations in *Previous_state* (S) and other states up to *Home_state* (S) should be stopped, and the value of e should not be defined. End of computations in *Home_state* (S) allows to define the value of the expression *Current_expression* (*Previous_state* (*Home_state* (S))) in the state *Previous_state* (*Home_state* (S)). It has been assumed that if the value of an expression is a result of an execution of a return statement, then it is equal to the value of this statement, which is stored in Last_value.

5.7 Correctness of $\mathcal{R}_0$

Below we give a series of theorems, which establish correctness of the rule $\mathcal{R}_0$. All these theorems are easy to prove, therefore we give only sketches of proofs or leave theorems without any proof.

As we have stated in Section 3.2, in this thesis we consider only rules, which we call *programs*, as defined by Definition 3.1. In such rules each variable var occurs in scope of rule **extend**:

extend UNIVERSE **with** var
    R_1
endextend

Theorem 5.1 *The rule $\mathcal{R}_0$ is a program.*

The next theorem deals with clashes of update rules. By a *clash* we call a situation in which two update rules

$$f(t_1,t_2,\ldots,t_n) := t$$
$$f(s_1,s_2,\ldots,s_n) := s$$

are executed in the same step in some algebra A, and for each i such that $1 \leq i \leq n$, t_i evaluates in A to the same value as s_i. If such a situation occurs, then the value of the updated term $f(t_1,t_2,\ldots,t_n)$ is not determined uniquely. However, $\mathcal{R}_0$ is defined in such a way, that a clash never occurs. Recall, that for a given algebra A and a transition rule R $E_A(R)$ denotes a set of applicable rules in according to Definition 3.2.

Theorem 5.2 *For each A the set $E_A(\mathcal{R}_0)$ does not contain two different update rules*

$$f(t_1,t_2,\ldots,t_n) := t$$

and

$$g(s_1, s_2, \ldots, s_n) := s$$

such that the function symbol f is equal to g.

In Appendix B we give a set $\mathcal{C}$ of constrains. All these constraints are satisfied by functions and constants in $\mathcal{A}_0$, i.e. if a constraint

$$f : \text{UNIVERSE}_1 \times \text{UNIVERSE}_2 \times \ldots \times \text{UNIVERSE}_n \rightarrow \text{UNIVERSE}$$

is in $\mathcal{C}$, then a function f is defined only for arguments $a_1, a_2, \ldots, a_n$ belonging to universes UNIVERSE_1, UNIVERSE_2, ..., UNIVERSE_n, respectively. Moreover, each value of f belongs to UNIVERSE. The same considerations apply to constants. Notice, that $\mathcal{C}$ consists of constraints for all constants and functions of $\mathcal{A}_0$.

For a given algebra $\mathcal{A}$ and a term t, a set (universe) of possible values of t can be determined from a given set of constraints. We will call this set a *type* of t. However, if a term t consists of variables, then interpretation of t can be given only if a rule $\mathcal{R}$ and an occurrence of t in $\mathcal{R}$ is given. Therefore for a given set of constraints $\mathcal{C}$ and a rule $\mathcal{R}$ we define a notion of a *type* of an occurrence of a term t in $\mathcal{R}$ as follows:

- if a term t is a constant c, and a constraint

  $$c : \text{UNIVERSE}$$

  is given in $\mathcal{C}$, then a type of each occurrence of t is UNIVERSE;

- if a term t is a variable v, and v occurs in a rule $\mathcal{R}'$ surrounded by **extend** rule of the form

  extend UNIVERSE **with** v
  $\quad\mathcal{R}'$
  endextend

  then this occurrence of t has a type UNIVERSE;

- if an occurrence of a term t is of the form $f(t_1, t_2, \ldots, t_n)$ and
  - terms $t_1, t_2, \ldots, t_n$ have types UNIVERSE_1, UNIVERSE_2, ..., UNIVERSE_n, respectively;
  - a constraint

    $$f : \text{UNIVERSE}_1 \times \text{UNIVERSE}_2 \times \ldots \times \text{UNIVERSE}_n \rightarrow \text{UNIVERSE}$$

    is given in $\mathcal{C}$;

  then an occurrence of t has a type UNIVERSE.

Since there is no nested **extend** rules in $\mathcal{R}_0$, each occurrence of each variable in $\mathcal{R}_0$ has a unique type. Moreover, for each constant and function symbol occurring in $\mathcal{R}_0$ exactly one constraint is given[2]. So, it is easy to verify the following lemma holds:

Lemma 5.1 *Each occurrence of each term in $\mathcal{R}_0$ has a type, and the type is unique.*

Notice, that a type of a ground term depends only on given set of constraints. Hence the definition:

Definition 5.1 *An algebra $\mathcal{A}$ satisfies given set of constraints if and only if for each ground term t which have a type* UNIVERSE, *a value of t belongs to* UNIVERSE.

Now we are ready to formulate the following theorem.

Theorem 5.3 *Let $\mathcal{A}_i$ be an algebra such that $\mathcal{A}_i$ satisfies the set $\mathcal{C}$ of constraints. If an algebra $\mathcal{A}_{i+1}$ is a result of applying the rule $\mathcal{R}_0$ to $\mathcal{A}_i$, then $\mathcal{A}_{i+1}$ satisfies $\mathcal{C}$.*

Proof
We prove, that no update rule changes satisfability of $\mathcal{C}$. Consider any update rule from $\mathcal{R}_0$:

$$t := s$$

By Lemma 5.1, t and s have types. It is easy to verify, that for each update rule in $\mathcal{R}_0$ a type of the left term t, say UNIVERSE, is equal to a type of the right term s. By assumption on $\mathcal{A}_i$, a value v of the term s in $\mathcal{A}_i$ belongs to UNIVERSE. So, v belongs to the type of t. Consequently a new value of t belongs to UNIVERSE.

■

5.8 Properties of the semantics

In this section we prove some facts, which justify correctness of our semantics. First we formalize some basic notions (as computation or order of computation), and then we prove properties of them.

We say that a sequence of algebras

$$\mathcal{R}_0^*(\mathcal{A}_0) = \mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i, \mathcal{A}_{i+1}, \ldots$$

is a *computation* if $\mathcal{A}_0$ is a initial state of our algebra as defined in Section 4.5.1, and $\mathcal{A}_{i+1}$ is an algebra obtained from $\mathcal{A}_i$ by applying the rule $\mathcal{R}_0$.

If two ground terms t_1 and t_2 are equal in some algebra $\mathcal{A}$, then we write

$$\mathcal{A} \models t_1 = t_2$$

[2]Some function symbols are used more than ones, but they have different definitions (constraints) and are considered as different symbols.

For elements of the universe EXPRESSIONS we define a relation $\prec$ which expresses an order of computation, i.e. expresses, that one expression is evaluated before another expression. We compare only expressions of the same method or the same block. We also consider only expressions evaluated in the same state. It implies that expressions in block which is defined in some method and expressions of the same method not included in this block are incomparable. Smalltalk in this point differs from other object–oriented languages like C++ or Java, because in Smalltalk an order of evaluation of all expressions are fixed, and each expression is evaluated at most once in one call of method or block, like in ML. We will prove, that our semantics satisfies these properties and it allows us to simplify some details. But first for expressions $e_1, e_2 \in$ EXPRESSIONS belonging to the same method $\mathcal{M}$ we define a relation $\prec$ which depends on a computation $\mathcal{R}_{\emptyset}^*(\mathcal{A}_0)$ and a state S:

$$\mathcal{R}_{\emptyset}^*(\mathcal{A}_0), S \models e_1 \prec e_2$$

if and only if there exist two algebras $\mathcal{A}_i$ and $\mathcal{A}_{i+k}$ in the computation $\mathcal{R}_{\emptyset}^*(\mathcal{A}_0)$ such that S exists in $\mathcal{A}_i$ and $\mathcal{A}_{i+k}$ and

$$\mathcal{A}_i \models (Expression\_value(e_1, S) \neq \mathsf{undef}) \wedge (Expression\_value(e_2, S) = \mathsf{undef})$$

and

$$\mathcal{A}_{i+k} \models (Expression\_value(e_1, S) \neq \mathsf{undef}) \wedge (Expression\_value(e_2, S) \neq \mathsf{undef})$$

where $k > 0$.

It is possible, that two expressions defined in the same method, are not in the relation $\prec$. It is shown by the following example

Example 5.1

```
sampleMethod
| rec |
rec messg: ([↑5] value).
```

The entire expression '`rec messg: ([↑5] value)`' is never evaluated, because its subexpression '`[↑5] value`' always causes escaping from the method, hence corresponding elements of the universe EXPRESSIONS are not in the relation $\prec$.

We show here some simple lemmas about any computation.

Lemma 5.2 *If $e \in$ EXPRESSIONS represents a message expression, then for each $S \in$ STATEMENTS, for each computation $\mathcal{R}_{\emptyset}^*(\mathcal{A}_0)$ such that there exists an algebra $\mathcal{A}$ in $\mathcal{R}_{\emptyset}^*(\mathcal{A}_0)$, such that*

$$\mathcal{A} \models Expression\_value(e, S) \neq \mathsf{undef}$$

the following facts hold:

- $\mathcal{R}_{\emptyset}^*(\mathcal{A}_0), S \models Receiver(e) \prec e$;

- *if First_argument(e)* $\neq$ undef *then*

$$\mathcal{R}_0^*(\mathcal{A}_0), S \models Receiver(e) \prec First\_argument(e);$$

- *if First_argument(e)* $\neq$ undef *then*

$$\mathcal{R}_0^*(\mathcal{A}_0), S \models First\_argument(e) \prec e;$$

- *if Next_argument(First_argument(e))* $\neq$ undef *then*

$$\mathcal{R}_0^*(\mathcal{A}_0), S \models First\_argument(e) \prec Next\_argument(First\_argument(e));$$

- *for all* $i \geq 1$ *such that Next_argument* $^{(i)}$*(First_argument(e))* $\neq$ undef *and Next_argument* $^{(i+1)}$*(First_argument(e))* $\neq$ undef

  $\mathcal{R}_0^*(\mathcal{A}_0), S \models$
  $$Next\_argument^{(i)}(First\_argument(e)) \prec Next\_argument^{(i+1)}(First\_argument(e));$$

- *for all* $i \geq 1$ *such that Next_argument* $^{(i)}$*(First_argument(e))* $\neq$ undef

$$\mathcal{R}_0^*(\mathcal{A}_0), S \models Next\_argument^{(i)}(First\_argument(e)) \prec e.$$

Proof
All these facts simply follow by an analysis of the rules EXPRESSION_EVALUATION and STATEMENT_EXECUTION.

■

Now we define a notion of subexpression in terms of our algebra $\mathcal{A}_0$. The definition is recursive. Let e be any element of the universe EXPRESSIONS. We say that e' is a *subexpression* of e if

- e' is a value of one of the terms: *Receiver*(e), *First_argument*(e) or *Next_argument* $^{(i)}$(*First_argument*(e)) or
- e' is a subexpression of *Receiver*(e), *First_argument*(e) or *Next_argument* $^{(i)}$(*First_argument*(e)).

The set of all subexpressions of the given e will be denoted as a *Sub*(e). For each $e' \in Sub(e)$ we can define a *path* from e to e', which is the following sequence of the functions

$$f_1 \circ f_2 \circ \ldots \circ f_n$$

such that

$$e' = f_n(f_{n-1}(\ldots f_1(e)\ldots))$$

where each f_i denotes either *Receiver*, either *First_argument* or *Next_argument*.

Lemma 5.3 *For each computation $\mathcal{R}_0^*(\mathcal{A}_0)$, a state S, $e \in$ EXPRESSIONS, and for each $e' \in Sub(e)$*

$$\mathcal{R}_0^*(\mathcal{A}_0), S \models e' \prec e.$$

Proof
The proof is by induction on the structure of the expression. If e is a constant, variable or block, then $Sub(e)$ is empty and the lemma holds. Inductive step concerns message expression. We show the proof for message expression e, which has one argument. It implies that

$$\forall e' \in Sub(Receiver(e)) : \ \mathcal{R}_0^*(\mathcal{A}_0), S \models e' \prec Receiver(e)$$

$$\forall e' \in Sub(First\_argument(e)) : \ \mathcal{R}_0^*(\mathcal{A}_0), S \models e' \prec First\_argument(e).$$

Of course,

$$Sub(Receiver(e)) \cup \{Receiver(e)\} \cup Sub(First\_argument(e)) \cup \{First\_argument(e)\} = Sub(e)$$

Then from Lemma 5.2 we obtain, that

$$\forall e' \in Sub(e) : \mathcal{R}_0^*(\mathcal{A}_0), S \models e' \prec e$$

The proof can be easily extended to the all message expressions with more than one argument or without any argument.

■

Now, we shall prove, that if we evaluate all subexpressions of some expression e, then we evaluate e just after all elements of $Sub(e)$ are evaluated.

Lemma 5.4 *For each computation $\mathcal{R}_0^*(\mathcal{A}_0)$, for each state S, for each $e \in$ EXPRESSIONS if there exists an algebra $\mathcal{A}_k$ in $\mathcal{R}_0^*(\mathcal{A}_0)$ such that $\mathcal{A}_k \models Expression\_value(e,S) \neq \mathsf{undef}$, then there is no such $e' \in$ EXPRESSIONS $\setminus Sub(e)$ such that*

$$\begin{aligned}&\forall f \in Sub(e) : \\ &\quad \mathcal{A}_i \models (Expression\_value(f,S) \neq \mathsf{undef}) \wedge (Expression\_value(e',S) = \mathsf{undef})\end{aligned}$$

and

$$\mathcal{A}_j \models Expression\_value(e',S) \neq \mathsf{undef}$$

where $i < j < k$.

Proof
The proof simply follows from the analysis of the rule EXPRESSION_EVALUATION and METHOD_LOOKUP.

■

As a simple corollary we obtain the following lemma

Lemma 5.5 *Let $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0)$ be any computation, let $\mathcal{A}_i$ be an element of $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0)$, and S be an element of the set* STATES *in $\mathcal{A}_i$. Let $e_1, e_2 \in$* EXPRESSIONS *represent expressions belonging to the same method, such that*

$$Sub(e_1) \cap Sub(e_2) = \emptyset$$

and

$$\mathcal{A}_i \models (Expression\_value\,(e_1, S) \neq \mathsf{undef}) \wedge (Expression\_value\,(e_2, S) \neq \mathsf{undef}).$$

Then either

$$\forall f_1 \in Sub(e_1) \cup \{e_1\}: \quad \forall f_2 \in Sub(e_2) \cup \{e_2\}: \quad \mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0), S \models f_1 \prec f_2$$

or

$$\forall f_1 \in Sub(e_1) \cup \{e_1\}: \quad \forall f_2 \in Sub(e_2) \cup \{e_2\}: \quad \mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0), S \models f_2 \prec f_1$$

i.e. all elements of $Sub(e_1) \cup \{e_1\}$ are evaluated before any element of $Sub(e_2) \cup \{e_2\}$ is evaluated, or each element of $Sub(e_2) \cup \{e_2\}$ is evaluated before $Sub(e_1) \cup \{e_1\}$.

Next lemma states, that the order of evaluation of expressions in a method or a block is fixed.

Lemma 5.6 *For each computation $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0)$, a state S, an expression e, and $e', e'' \in Sub(e)$ if*

$$\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0), S \models e' \prec e''$$

then for all computation $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0')$ and state S'

$$\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0'), S' \models e'' \not\prec e'$$

Proof
We prove it by contradiction. Suppose that there exists a computation $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0')$ and a state S' that

$$\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0'), S' \models e'' \prec e'$$

We consider three cases depending on the path from e to e' and e''.

If e' lies on the path from e to e'' then by Lemma 5.2 for all $\mathcal{R}_{\emptyset}^{*}(\mathcal{A}_0), S \models e' \prec e''$, which gives a contradiction.

In the second case: e'' lies on the path from e to e', then we use the same argument to show, that this case is not possible.

In the third case there is a common prefix of the paths from e to e' and from e to e''. Let $f_1 \circ f_2 \circ \ldots \circ f_i \circ g_1 \circ g_2 \ldots g_m$ be a path from e to e' and $f_1 \circ f_2 \circ \ldots \circ f_i \circ h_1 \circ h_2 \ldots h_n$ be a path from e to e'', where g_1 is different from h_1. Let

$$\overline{e} = f_i(f_{i-1}(\ldots f1(\overline{e})\ldots))$$

It is easy to notice, that

$$e' \in Sub(g_1(\overline{e})) \cup \{g_1(\overline{e})\}$$

and

$$e'' \in Sub(h_1(\overline{e})) \cup \{h_1(\overline{e})\}$$

So, by Lemma 5.5, if there exists a computation $\mathcal{R}_\Theta^*(\mathcal{A}_0)$ and a state S such that

$$\mathcal{R}_\Theta^*(\mathcal{A}_0), S \models e_1 \prec e_2$$

then there is no such a computation $\mathcal{R}_\Theta^*(\mathcal{A}_0')$ and state S such that

$$\mathcal{R}_\Theta^*(\mathcal{A}_0)', S' \models e_2 \prec e_1$$

■

Therefore from now on if we talk about relation $\prec$ we may omit computation and state.

5.9 An example of computation

In this section we give a complete semantics describing computations of two simple methods. We assume that a class ***True*** is declared in Smalltalk, with two methods *'test'* and *'ifTrue:ifFalse:'* where

Example 5.2

```
test
self ifTrue: [↑1] ifFalse: [↑0]
```

and

```
ifTrue: trueBlock ifFalse: falseBlock
↑trueBlock value
```

First we give a definition of the algebra $\mathcal{A}_0$ corresponding to this example. We assume, that there is an element True in $\mathcal{A}_0$, which corresponds to the class ***True*** and belongs to the universe CLASSES. The method *'test'* will be represented in $\mathcal{A}_0$ by an element M_{test}, which belongs to METHODS, and the method *'ifTrue:ifFalse:'* will be represented by M_{if}, also belonging to METHODS. We assume that no shared variable is used by methods of ***True***, therefore we put

Shared_pools (True) = empty

Expressions of the method *'test'* will be represented by the set of elements

$\mathcal{T}$ = { t, t_{self}, $\mathsf{t}_{[\uparrow 1]}$, $\mathsf{t}_{\uparrow 1}$, $\mathsf{t}_{[\uparrow 0]}$, $\mathsf{t}_{\uparrow 0}$ }

which belong to the universe EXPRESSIONS. The correspondence among these elements and expressions of *'test'* is the following:

- t corresponds to `'self ifTrue: [↑1] ifFalse: [↑0]'`;
- t_{self} corresponds to `'self'`;

- $\mathsf{t}_{\texttt{[}\uparrow\texttt{1]}}$ corresponds to `'[↑1]'`;
- $\mathsf{t}_{\uparrow\texttt{1}}$ corresponds to `'↑1'`;
- $\mathsf{t}_{\texttt{[}\uparrow\texttt{0]}}$ corresponds to `'[↑0]'`;
- $\mathsf{t}_{\uparrow\texttt{0}}$ corresponds to `'↑0'`.

Below we give a list of equalities satisfied in $\mathcal{A}_0$, which describes the method *'test'*. In order to increase readability, we divide them into groups.

The first group of equalities concerns the element M_{test}.

$$\begin{aligned}
&\textit{Method}\,(\textit{'test'}, \mathsf{True}) = \mathsf{M}_{test}\\
&\textit{Method_class}\,(\mathsf{M}_{test}) = \mathsf{True}\\
&\textit{Argument_names}\,(\mathsf{M}_{test}) = \mathsf{empty}\\
&\textit{Temporary_names}\,(\mathsf{M}_{test}) = \mathsf{empty}
\end{aligned}$$

The equalities given below represent a structure of the method `'true'`, i.e. relationships among elements $\mathcal{T}$.

$$\begin{aligned}
&\textit{First_statement}\,(\mathsf{M}_{test}) = \mathsf{t}\\
&\textit{Name}\,(\mathsf{t}) = \textit{'ifTrue:ifFalse:'}\\
&\textit{Receiver}\,(\mathsf{t}) = \mathsf{t}_{self}\\
&\textit{First_argument}\,(\mathsf{t}) = \mathsf{t}_{\texttt{[}\uparrow\texttt{1]}}\\
&\textit{Next_argument}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{1]}}) = \mathsf{t}_{\texttt{[}\uparrow\texttt{0]}}\\
&\textit{Superexpression}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{0]}}) = \mathsf{t}
\end{aligned}$$

The next group of equalities gives more details on elements of $\mathcal{T}$.

$$\textit{Name}\,(\mathsf{t}) = \textit{'ifTrue:ifFalse:'}$$

$$\begin{aligned}
&\textit{Name}\,(\mathsf{t}_{self}) = \textit{'self'}\\
&\textit{Superexpression}\,(\mathsf{t}_{self}) = \mathsf{t}
\end{aligned}$$

$$\begin{aligned}
&\textit{First_statement}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{1]}}) = \mathsf{t}_{\uparrow\texttt{1}}\\
&\textit{Argument_names}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{1]}}) = \mathsf{empty}\\
&\textit{Constant_value}\,(\mathsf{t}_{\uparrow\texttt{1}}) = 1\\
&\textit{Return_prefix}\,(\mathsf{t}_{\uparrow\texttt{1}}) = \text{'}\uparrow\text{'}
\end{aligned}$$

$$\begin{aligned}
&\textit{First_statement}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{0]}}) = \mathsf{t}_{\uparrow\texttt{0}}\\
&\textit{Argument_names}\,(\mathsf{t}_{\texttt{[}\uparrow\texttt{0]}}) = \mathsf{empty}\\
&\textit{Constant_value}\,(\mathsf{t}_{\uparrow\texttt{0}}) = 0\\
&\textit{Return_prefix}\,(\mathsf{t}_{\uparrow\texttt{1}}) = \text{'}\uparrow\text{'}
\end{aligned}$$

Expressions of the method *'ifTrue:ifFalse:'* are represented by elements

$$\mathcal{F} = \{\,\mathsf{f}, \mathsf{f}_{\texttt{trueBlock}}\}$$

belonging to EXPRESSIONS. They correspond to the expressions `'trueBlock value'` and `'trueBlock'`, respectively. The following list of equalities defines the method *'ifTrue:ifFalse:'*:

Method (*'ifTrue:ifFalse:'*, True) = M_{if}
Method_class (M_{if}) = True
Argument_names (M_{if}) = [*'trueBlock' 'falseBlock'*]
Temporary_names (M_{if}) = empty

First_argument (M_{if}) = f
Receiver (f) = $\mathsf{f}_{\texttt{trueBlock}}$
Superexpression ($\mathsf{f}_{\texttt{trueBlock}}$) = f

Name (f) = *'value'*
Name ($\mathsf{f}_{\texttt{trueBlock}}$) = *'trueBlock'*

The assignment statement never occurs in these methods, therefore the value of the function *Assignments* is empty for each element of $\mathcal{T}$ and $\mathcal{F}$. We assume, that the value of a variable *'true'* is an object belonging to the class ***True***. Hence the expression

```
true test
```

is valid and begins a computation. The object, which is the value of *'true'*, will be denoted in $\mathcal{A}_0$ by an element true.

Below we describe a computation $\mathcal{R}_0^*(\mathcal{A}_0)$ which corresponds to an evaluation of the expression `'true test'` in Smalltalk. The computation $\mathcal{R}_0^*(\mathcal{A}_0)$ is a sequence of algebras

$$\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_{29}$$

where each $\mathcal{A}_{i+1}$ is a result of applying the rule $\mathcal{R}_0$ to the algebra $\mathcal{A}_i$. For each algebra $\mathcal{A}_i$ we give a rule (or rules) by which this algebra has been obtained. Recent changes of algebra are also presented in form of equalities.

$\mathcal{A}_0$: In order to begin a computation, the constants Receiver and Method should be set (see Section 4.5.5).

Receiver = true
Method = M_{test}

$\mathcal{A}_1$: METHOD_STATE_CREATION and NEW_METHOD_STATE

At this and the next stage the state S_1 is created and initialized. In this state M_{test} will be computed.

Argument_names = empty
Temporary_names = empty
Argument = undef
Previous_state (S_1) = undef
Current_state = S_1
Method_class (S_1) = True
Current_expression (S_1) = t
Home_state (S_1) = S_1

$\mathcal{A}_2$: METHOD_STATE_CREATION

Argument_names = undef
Method = undef
Receiver = Receiver

$\mathcal{A}_3$: EXPRESSION_EVALUATION

In order to evaluate the message expression `'self ifTrue: [↑1] ifFalse: [↑0]'`, the receiver represented here by t_{self} is evaluated.

$Current\_expression\,(S_1) = t_{self}$

$\mathcal{A}_4$: VARIABLE_DEREFERENCE

At this point the value of t_{self} is evaluated. The constant Assignments is set to empty. Notice, that no value is assigned by an assignment statement '$\leftarrow$ ', therefore Assignments is set to empty.

$Expression\_value\,(t_{self}, S_1) = \mathsf{true}$
Assignments = empty

$\mathcal{A}_5$: STATEMENT_EXECUTION

Now the computation is moved back to t, but t is still not evaluated.

$Current\_expression\,(S_1) = t$

$\mathcal{A}_6$: EXPRESSION_EVALUATION

The receiver of t is already evaluated, but arguments of t are not evaluated, hence computation is moved to $t_{[\uparrow 1]}$, the first argument of t.

$Current\_expression\,(S_1) = t_{[\uparrow 1]}$

$\mathcal{A}_7$: BLOCK_CREATION

At this stage the block expression `'[↑1]'` is evaluated. It causes creation of a new block. In algebra it is reflected by creation of a new element of the universe OBJECTS. This element will be denoted by a symbol Block_1.

$Expression\_value\,(t_{[\uparrow 1]}, S_1) = \mathsf{Block}_1$
$Class\,(\mathsf{Block}_1) = \mathsf{block}$
$Home\_state\,(\mathsf{Block}_1) = S_1$
$First\_statement\,(\mathsf{Block}_1) = t_{\uparrow 1}$
$Argument\_names\,(\mathsf{Block}_1) = \mathsf{empty}$
Assignments = empty

$\mathcal{A}_8$: STATEMENT_EXECUTION

After evaluation of the first argument, $t_{[\uparrow 1]}$, the second argument, `'[↑0]'`, will be evaluated.

$Current\_expression\,(S_1) = t_{[\uparrow 0]}$

$\mathcal{A}_9$: BLOCK_CREATION

As in the algebra $\mathcal{A}_7$, a new element representing a block object, is created and is denoted by a symbol Block_2.

Expression_value ($t_{[\uparrow 0]}$, S_1) = $Block_2$
Class ($Block_2$) = block
Home_state ($Block_2$) = S_1
First_statement ($Block_2$) = $t_{\uparrow 0}$
Argument_names ($Block_2$) = empty
Assignments = empty

$\mathcal{A}_{10}$: STATEMENT_EXECUTION

After computing the receiver and arguments of t, the computation moves again to t.

Current_expression (S_1) = t

$\mathcal{A}_{11}$: INITIAL_CLASS

Since t represents a message expression, then method (an element representing a method) should be found.

Class = True

$\mathcal{A}_{12}$: METHOD_LOOKUP

Since the method of the signature *'iftrue:ifFalse:'* is defined in the class ***True***, the element M_{if}, corresponding to the requested method, is found in one step.

Method = M_{if}
Receiver = true
Class = undef

$\mathcal{A}_{13}$: METHOD_STATE_CREATION and NEW_METHOD_STATE

For the element M_{if}, a new state S_2 is created and S_2 is initialized.

Previous_state (S_2) = S_1
Current_state = S_2
Method_class (S_2) = True
Receiver (S_2) = true
Current_expression (S_2) = f
Home_state (S_2) = S_2
Argument_names = [*'trueBlock' 'falseBlock'*]
Temporary_names = empty
Argument = $t_{[\uparrow 1]}$

$\mathcal{A}_{14}$: METHOD_STATE_CREATION and METHOD_ARGUMENT_PASSING

Here the first argument of the method being called is initialized and a new variable, denoted by Var_1, corresponding to this argument is created.

Variable_value (Var_1) = $Block_1$
Temporary_variable (*'trueBlock'*, S_2) = Var_1
Argument_names = [*'falseBlock'*]
Argument = $t_{[\uparrow 0]}$

$\mathcal{A}_{15}$: METHOD_STATE_CREATION and METHOD_ARGUMENT_PASSING

Here the variable denoted by Var_2, corresponding to the second argument, is created and initialized.

Variable_value (Var_2) = Block_2
Temporary_variable (*'falseBlock'*, S_2) = Var_2
Argument_names = empty
Argument = undef

$\mathcal{A}_{16}$: METHOD_STATE_CREATION

Here the final changes concerning creation of a method state are given.

Argument_names = undef
Method = undef
Receiver = undef

$\mathcal{A}_{17}$: EXPRESSION_EVALUATION

Since to evaluate f the value of f is needed, the computation moves to $\mathsf{f}_{\mathtt{trueBlock}}$.

Current_expression (S_2) = $\mathsf{f}_{\mathtt{trueBlock}}$

$\mathcal{A}_{18}$: VARIABLE_DEREFERENCE

$\mathsf{f}_{\mathtt{trueBlock}}$ represents a reference to a variable. Therefore $\mathcal{A}_{18}$, $\mathcal{A}_{19}$ and $\mathcal{A}_{20}$ represent a computation of a private variable named *'trueBlock'*.

Variable_name = *'trueBlock'*
Pool_names = empty
Class = True

$\mathcal{A}_{19}$: PRIVATE_VARIABLE

Variable = Var_1

$\mathcal{A}_{20}$: VARIABLE_DEREFERENCE

Expression_value ($\mathsf{f}_{\mathtt{trueBlock}}$, S_2) = Block_2
Assignments = empty
Variable = undef
Variable_name = undef

$\mathcal{A}_{21}$: STATEMENT_EXECUTION

$\mathsf{f}_{\mathtt{trueBlock}}$, which represents a receiver of the message expression `'trueBlock value'`, is already evaluated. Now a method corresponding to the class of the receiver of t and the signature *'value'* should be found.

Current_expression (S_2) = f

$\mathcal{A}_{22}$: INITIAL_CLASS

Class = block

$\mathcal{A}_{23}$: INITIAL_CLASS

By the rule METHOD_LOOKUP an element which represents a method *'value'* is found. The method *'value'* evaluates the body of the block.

> Method = *Method* (*'value'*, block)
> Receiver = Block_1
> Class = undef

$\mathcal{A}_{24}$: BLOCK_STATE_CREATION and NEW_BLOCK_STATE

At this and the next stage a state S_3 for computing the block Block_1 body. Notice that the *Home_state* (S_3) is equal to S_1, and in S_1 Block_1 has been created.

> *Home_state* (S_3) = S_1
> *Current_expression* (S_3) = $t_{\uparrow 1}$
> *Previous_state* (S_3) = S_2
> Current_state = S_3
> Argument_names = empty
> Argument = undef

$\mathcal{A}_{25}$: BLOCK_STATE_CREATION

> Argument_names = undef
> Method = undef
> Receiver = undef
> Last_value = nil

$\mathcal{A}_{26}$: PRIMARY_EVALUATION

At this stage the computation has reached the last expression of the body of Block_1. The next three stages describe preparation for return from Block_1.

> *Expression_value* ($t_{\uparrow 1}$, S_3) = 1
> Assignments = empty

$\mathcal{A}_{27}$: STATEMENT_EXECUTION

> Last_value = 1
> Return_prefix = '$\uparrow$'
> *Current_expression* (S_3) = undef

$\mathcal{A}_{28}$: SETTING_SUPERFLUOUS_STATE

Since the home state of Block_1 is S_1, return statement '$\uparrow$1' is interpreted as return from S_1. However, *Previous_state* (S_1) is equal to undef, therefore the state, where computation should be continued, is set to undef in $\mathcal{A}_{28}$.

> Superfluous_state = S_1

$\mathcal{A}_{29}$: COMPLETION

In algebra $\mathcal{A}_{29}$, no changes can be introduced by the rule $\mathcal{R}_0$. Therefore computation is completed.

> Current_state = undef

Conclusions

Our semantics is complete in the sense that it defines the meaning of computation in Smalltalk. Of course, it is not a full description of Smalltalk. We omit, for example, objects creation treating it as an execution of a method *'new'* in each class. Classes are distinguished objects, which can create other objects. To keep our model compact we have also skip arithmetic, input/output, and other features. Moreover, extension for standard classes mentioned in [ANS97] requires at least introduction of operations allowing access to hardware (e.g. input/output) or arithmetic.

Chapter 6

Continuation model of Smalltalk operational semantics

6.1 Introduction to continuation

Continuation is a concept in semantics of programming languages which expresses explicitly control flow of program. Let us consider a sequence of statements of some imperative programs (say Pascal):

$$S_1; S_2; \ldots S_n$$

The standard meaning of this sequence is the following:

> *if a statement S_i has been computed and S_i does not include any jump statement, then computation is continued by computing S_{i+1}.*

Such a rule is directly and elegantly expressed in operational or denotational semantics, because both semantics need only S_i and S_{i+1}. If S_i includes jump, then semantics becomes complicated since semantics of S_i strongly depends on the entire program, what raises many problems with description of the program behavior. The solution of this problem is a concept of *continuation*, where next point of computation is expressed as an abstract domain ([Gor79]) or a function (e.g. escapes in [Rey81]).

One interesting application of continuation is presented by A. Appel in [App92]. He used a language based on continuation–passing style as an intermediate language representing the source program which fully describes semantics and compiler of Standard ML.

The concept of continuation is also implemented in programming languages (mainly in functional languages, as Scheme or ML) as a primitive called *call with current continuation* (call/cc). It is used for implementing breakpoints or concurrent processes, escaping from loop, etc. In the last application functionality of continuation is similar to `goto` statement, but is more abstract.

Notice, that an order, in which expressions are evaluated, is not explicitly given in the semantics presented so far. In this chapter we shall give slightly reformulated definition of semantics, which directly expresses an order of computation.

In order to provide some intuitions, let us consider again Example 5.2 of computation presented in Section 5.9. In this example an expression `'self ifTrue: [↑1] ifFalse: [↑0]'` (denoted here by *exp*) is represented by a set $\mathcal{T}$ of elements { t, t_{self}, $\mathsf{t}_{[\uparrow 1]}$, $\mathsf{t}_{\uparrow 1}$, $\mathsf{t}_{[\uparrow 0]}$, $\mathsf{t}_{\uparrow 0}$} belonging to EXPRESSIONS. During computation *exp* is evaluated in a state S_1. Consider the term *Current_expression* (S_1) representing expression being currently evaluated. *Current_expression* (S_1) assumes the following values t, t_{self}, t, $\mathsf{t}_{[\uparrow 1]}$, $\mathsf{t}_{[\uparrow 0]}$, t in algebras $\mathcal{A}_1$, $\mathcal{A}_3$, $\mathcal{A}_5$, $\mathcal{A}_6$, $\mathcal{A}_7$ and $\mathcal{A}_{10}$, respectively. The order of evaluation is shown in Figure 6.1, where arcs point to the next value of *Current_expression* (S_1).

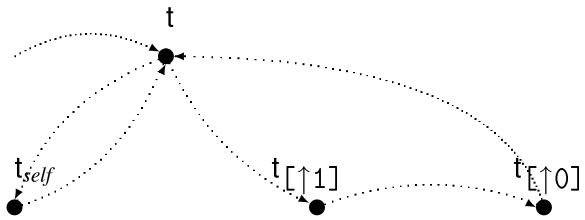

Figure 6.1: Evaluation using the initial algebra $\mathcal{A}_0$ and the rule $\mathcal{R}_0$

It is easy to notice, that t becomes the value of *Current_expression* (S_1) three times: in $\mathcal{A}_1$, $\mathcal{A}_5$ and $\mathcal{A}_{10}$, but only in $\mathcal{A}_{10}$ evaluation of t begins. So, the stages $\mathcal{A}_1$ and $\mathcal{A}_5$ seem to be superfluous. In the case of more complicated expressions, much more of there may be much more such superfluous stages. Observe, that if an element $e \in$ EXPRESSIONS represents a message expression, then e is 'visited' three times in a given state, where only the last 'visit' causes evaluation of e. In the case of message expression with no argument, e is visited only twice.

So, the order in which elements of $\mathcal{T}$ are evaluated is the following:

t_{self} $\quad$ $\mathsf{t}_{[\uparrow 1]}$ $\quad$ $\mathsf{t}_{[\uparrow 0]}$ $\quad$ t

what we show in Figure 6.2 by arcs.

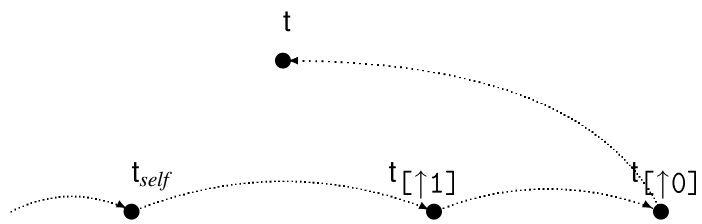

Figure 6.2: Other order of evaluation

In the next sections we shall define an algebra $\mathcal{B}_0$ and a rule $\mathcal{R}_1$ by slightly reformulation of the algebra $\mathcal{A}_0$ and the rule $\mathcal{R}_0$ given in chapters 4 and 5. We shall also define a function

Cont, which for a given $e \in$ EXPRESSIONS points directly the element e' of EXPRESSIONS to be evaluated in the next step. This function will corresponds to the arcs in Figure 6.2.

6.2 Definition of continuation

The signature of algebra $\mathcal{B}_0$ includes the signature of $\mathcal{A}_0$ and a function symbol

$$Cont : \text{EXPRESSIONS} \rightarrow \text{EXPRESSIONS}$$

which directly expresses an order of computation. For a given Smalltalk program P, all universes and functions of $\mathcal{B}_0$ are defined in the same way, as $\mathcal{A}_0$, as presented in Chapter 4, except two functions corresponding to the function symbol *First_statement*. Definition of the function *Cont* is given below. Definition of *First_statement* is given in Section 6.3.

First we consider an order of computation of some expression *exp* which contains at least one message expression. Suppose that an element $e \in$ EXPRESSIONS is defined for *exp*. We want to know which of the expressions in $\{e\} \cup Sub(e)$ is evaluated first in the given state S, where the function *Sub* is defined in Section 5.8. As we have proven in Section 5.8, before evaluating e in a state S, an element e' of $Sub(e)$ is evaluated. From Lemmas 5.2 and 5.3 we obtain, that e' is in $Sub(Receiver(e))$, or if $Sub(Receiver(e))$ is empty, then e' is equal to $Receiver(e)$. Of course, if $Sub(e)$ is empty, i.e. if e corresponds to a constant, variable or block, then only e is evaluated. Now, we define an auxiliary function

$$FirstExpr : \text{EXPRESSIONS} \rightarrow \text{EXPRESSIONS}$$

which indicates such an element e' for given e. *FirstExpr* is defined as follows:

$$FirstExpr(e) = \begin{cases} e & \text{if } Sub(e) \text{ is empty} \\ FirstExpr(Receiver(e)) & \text{otherwise} \end{cases}$$

Now, we are able to define the function *Cont*. Suppose that in a state $S \in$ STATES an $e \in$ EXPRESSIONS is just evaluated. According to the rule STATEMENT_EXECUTION (see Section 5.5) the pointer to the currently evaluated expression indicated by *CurrExp* is assigned to $Superexpression(e)$, $Next\_argument(e)$ or to $Next\_statement(e)$. By Lemma 4.2 we know that at most one of these expressions is different from undef. Suppose that one of this expression, $f(e)$, is defined. If f is *Next_argument* or *Next_statement*, then we evaluate $FirstExpr(f(e))$ first. Hence

- if $Next\_statement(e) \neq$ undef then we put

$$Cont(e) = FirstExpr(Next\_statement(e))$$

- if $Next\_argument(e) \neq$ undef then we put

$$Cont(e) = FirstExpr(Next\_argument(e))$$

If f equals to *Superexpression*, then *Superexpression* (e) must correspond to a message expression. Therefore analyzing the definition of the function *Superexpression* (see Section 4.3.3.5) we obtain two cases: e is the value of the term *Receiver* (*Superexpression* (e)), or e is the last argument of the *Superexpression* (e). Analyzing the rule STATEMENT_EXECUTION and the rule EXPRESSION_EVALUATION (see Sections 4.3.3.5 and 4.3.3.5). we define *Cont* as follows:

if $Receiver\,(Superexpression\,(e)) = e$ then

if $First\_argument\,(Superexpression\,(\mathrm{e})) \neq \mathsf{undef}$ then we put

$$Cont(e) = FirstExpr(First\_argument(Superexpression(e)))$$

else we put

$$Cont(e) = Superexpression(e)$$

else we put

$$Cont(e) = Superexpression(e)$$

As a claim we obtain the observation which states, that the function *Cont* constitutes an order equivalent to the relation $\prec$ defined in Section 5.8.

Proposition 6.0.1 *For each $e \in$ EXPRESSIONS if $Cont(e) \neq \mathsf{undef}$ and for each computation $C = \mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i, \mathcal{A}_{i+1}, \ldots$ if there exists an element S of the universe STATES such that*

$$\mathcal{A}_i \models Expression\_value(Cont(e), S) \neq \mathsf{undef}$$

then

$$C, S \models e \prec Cont(e)$$

6.3 The rule $\mathcal{R}_1$

Having the function *Cont* and the function *FirstExpr* in the algebra $\mathcal{R}_1$, we slightly modify the function *First_statement* defined on METHODS (see Section 4.2.4) and the function *First_statement* defined on EXPRESSIONS (see Section 4.3.3.3). For each $\mathsf{M} \in$ METHODS such that the equality

$$First\_statement\,(\mathsf{M}) = e$$

holds in $\mathcal{A}_0$, then in $\mathcal{B}_0$ we put

$$First\_statement\,(\mathsf{M}) = FirstExpr\,(e)$$

For each $\mathsf{e} \in$ EXPRESSIONS such that the equality

$$First\_statement\,(\mathsf{e}) = \mathsf{f}$$

holds in $\mathcal{A}_0$, then in $\mathcal{B}_0$ we put

$$First\_statement\,(\mathsf{e}) = FirstExpr\,(\mathsf{f})$$

The rule $\mathcal{R}_1$ is a modified version of the rule $\mathcal{R}_0$. $\mathcal{R}_1$ includes all rules included in $\mathcal{R}_0$, except the rules STATEMENT_EXECUTION and EXPRESSION_EVALUATION.

The rule STATEMENT_EXECUTION has now a simplified form, because three conditions including functions *Next_statement*, *Next_argument* and *Superexpression* are replaced by one condition including *Cont*. The definition of the function *Cont* and Observation 6.0.1 ensure,

---

Rule STATEMENT_EXECUTION
  if $\underline{CurrExpVal}$ = undef **then**
    EXPRESSION_EVALUATION
  elseif Assignments ≠ empty **then**
    ASSIGNMENT
  elseif *Cont* ($\underline{CurrExp}$) ≠ undef **then**
    $\underline{CurrExp}$:= *Cont* ($\underline{CurrExp}$)
  else
    Last_value := $\underline{CurrExpVal}$
    Return_prefix := *Return_prefix* ($\underline{CurrExp}$)
    $\underline{CurrExp}$:= undef
  endif
EndRule

---

that if an expression e_r is a receiver of a message expression e, then it is computed before e, since

$$Cont^{(i)}(e_r) = e$$

Hence it is not necessary to test a condition whether a receiver is already evaluated or not, as in STATEMENT_EXECUTION in $\mathcal{R}_0$. The same reasoning can be applied to the arguments of message expression. Therefore we simplify the rule EXPRESSION_EVALUATION to the given form.

Rule EXPRESSION_EVALUATION
  if *Name* ($\underline{CurrExp}$) = undef **then**
    PRIMARY_EVALUATION
  else
    METHOD_LOOKUP
  endif
EndRule

6.4 Example

Consider Example 5.2 of a method *'test'* and its representation in the algebra $\mathcal{A}_0$ given in Section 5.9. A new continuation representation of *'test'* in the algebra $\mathcal{B}_0$ can be obtained by the following modification of $\mathcal{A}_0$:

- The definition of *First_statement* is changed. In $\mathcal{B}_0$ we put

  $\mathit{First\_statement}\,(\mathsf{M}_{test}) = \mathsf{t}_{self}$

- In $\mathcal{B}_0$ a new function *Cont* is added. We put

  $\mathit{Cont}\,(\mathsf{t}_{self}) = \mathsf{t}_{[\uparrow 1]}$
  $\mathit{Cont}\,(\mathsf{t}_{[\uparrow 1]}) = \mathsf{t}_{[\uparrow 0]}$
  $\mathit{Cont}\,(\mathsf{t}_{[\uparrow 0]}) = \mathsf{t}$

  Notice, that these equalities correspond to arcs in Figure 6.2.

By applying $\mathcal{R}_1$ to $\mathcal{B}_0$, we obtain a sequence of algebras

$$\mathcal{R}_1^*(\mathcal{B}_0) = \mathcal{B}_0, \mathcal{B}_1, \ldots,$$

Let us compare the computation to the computation $\mathcal{R}_0^*(\mathcal{A}_0)$ given in Section 5.9. The algebra $\mathcal{B}_0$ is similar to $\mathcal{A}_0$ up to differences given above, and $\mathcal{B}_1$ is similar to $\mathcal{A}_1$, except that a value of $\mathit{Current\_expression}\,(\mathsf{S}_1)$ is t_{self}. In $\mathcal{B}_2$ t_{self} is already evaluated, as in $\mathcal{A}_3$. Notice, that $\mathcal{A}_2$ has no corresponding $\mathcal{B}_i$. Next, an equality $\mathit{Current\_expression}\,(\mathsf{S}_1) = \mathsf{t}_{[\uparrow 1]}$ is put yielding $\mathcal{B}_3$, what corresponds to $\mathcal{A}_5$.

The remaining steps of $\mathcal{R}_1^*(\mathcal{B}_0)$ computation are similar to the $\mathcal{R}_0^*(\mathcal{A}_0)$.

6.5 Equivalence of semantics

By now we have provided two ways of defining operational semantics of a given Smalltalk program P. These semantics are given as two Abstract State Machines: $\langle \mathcal{A}_0, \mathcal{R}_0 \rangle$ and $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$. In this section we define a notion of equivalence of algebras and show, that both ASM's give equivalent result.

Let P be a Smalltalk program. Let $\mathcal{A}_0$ be the algebra defined for P as defined in Chapter 4, and let $\mathcal{B}_0$ be the algebra defined for P according to the definition given in Chapter 6. $\mathcal{R}_0$ and $\mathcal{R}_1$ are transition rules defined in Chapter 5 and Chapter 6, respectively.

Universes and functions of $\mathcal{A}_0$ and $\mathcal{B}_0$ are defined in a very similar way. In particular, as defined in Section 6.2, $\mathcal{B}_0$ has the same universes and definition of elements in $\mathcal{A}_0$ and $\mathcal{B}_0$ are identical. So, we define equivalence between elements of $\mathcal{A}_0$ and $\mathcal{B}_0$ as follows:

Definition 6.1

$$a \equiv b$$

if and only if $a \in \mathcal{A}$ and $b \in \mathcal{B}$, and a and b are defined either for the same name, list of names, object, variable, method, expression or state in P.

Let $\Sigma_{\mathcal{A}}$ be the signature of $\mathcal{A}_0$, and $\Sigma_{\mathcal{B}}$ be the signature of $\mathcal{B}_0$. $\Sigma_{\mathcal{B}}$ is an extension of $\Sigma_{\mathcal{A}}$ by additional unary function symbol *Cont*. Except the functions corresponding to the function symbol *First_statement*, all functions in $\mathcal{A}_0$ and $\mathcal{B}_0$ are defined in the same way, as stated in Section 6.2. By Π we denote the signature obtained from $\Sigma_{\mathcal{A}}$ by omitting the function symbol *First_statement*.

We claim that

Proposition 6.0.2 *Let $f_{\mathcal{A}}$ be a function in the structure of $\mathcal{A}_0$, let $f_{\mathcal{B}}$ be a function in the structure of $\mathcal{B}_0$, and assume that $f_{\mathcal{A}}$ and $f_{\mathcal{B}}$ correspond to the same n–ary function symbol f in Π. For each $a_1,\ldots,a_n \in \mathcal{A}_0$, $b_1,\ldots,b_n \in \mathcal{B}_0$, we have:*

$$a_1 \equiv b_1,\ldots,a_n \equiv b_n \Rightarrow f_{\mathcal{A}}(a_1,\ldots,a_n) \equiv f_{\mathcal{B}}(b_1,\ldots,b_n) \tag{6.1}$$

If for each function symbol f from given signature Σ, equality (6.1) is satisfied in two algebras C and D, then we shall say, that C and D are equivalent up to Σ, and shall write it as follows:

$$\mathcal{C} \equiv_{\Sigma} \mathcal{D}$$

Of course, if two algebras C and D are equivalent up to Σ, then for each term t over Σ interpretations of t in C and D are equivalent.

By $R(C)$ we shall denote a result of applying a rule R to an algebra C.

It is also easy to notice, that applying the same update rule to equivalent algebras preserves equivalence. We state it formally:

Proposition 6.0.3 *Let C and D be algebras, and Σ be nonempty signature such that $C \equiv_{\Sigma} D$. Let R be a set of nonclashing update rules built over Σ. The following equivalence holds*

$$R(C) \equiv_{\Sigma} R(D)$$

An additional problem arises if a set R of update rules contains also the rule **extend**. We consider only the case, when R contains at most one extend rule.

Definition 6.2 *Let C and D be two algebras such that*

$$C \equiv_{\Sigma} D$$

Let R be a set of update rules over Σ including one extend rule. Let c and d be elements created by applying R to C and D, respectively. If we put

$$c \equiv d$$

then results of applying R to C and D are also equivalent up to Σ.

It implies, that if $C_0 \equiv_\Sigma D_0$ and R is a transition rule built over Σ, then for each $k > 0$

$$R^k(C_0) \equiv_\Sigma R^k(D_0)$$

In other words two computations

$$R^*(C_0) = C_0, C_1, C_2, \ldots, C_n, \ldots$$

and

$$R^*(D_0) = D_0, D_1, D_2, \ldots, D_n, \ldots$$

are equivalent up to Σ and give equivalent results.

Now we are going to consider equivalence of computation of two Abstract State Machines: $\langle \mathcal{A}_0, \mathcal{R}_0 \rangle$ and $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$.

It is easy to notice, that the following properties hold

- $\mathcal{A}_0 \equiv_\Pi \mathcal{B}_0$;
- for each algebra $\mathcal{A}$ the set of applicable rules $E_\mathcal{A}(\mathcal{R}_0)$ contains at most one rule **extend** (see Definition 3.2 in Section 3.2);
- for each algebra $\mathcal{B}$ the set of applicable rules $E_\mathcal{B}(\mathcal{R}_1)$ contains at most one rule **extend**.

Let us consider differences between $\mathcal{R}_0$ and $\mathcal{R}_1$. $\mathcal{R}_1$ differs from $\mathcal{R}_0$ on two subrules: STATEMENT_EXECUTION and EXPRESSION_EVALUATION. Moreover, differences concern only these update rules, which change a value of the function *Current_expression*. Notice, that *Current_expression* is changed only in STATEMENT_EXECUTION and EXPRESSION_EVALUATION in $\mathcal{R}_0$, and STATEMENT_EXECUTION in $\mathcal{R}_1$. The rule STATEMENT_EXECUTION (in $\mathcal{R}_0$ and $\mathcal{R}_1$) is applied if current expression has already been evaluated and its value has been assigned to variables. EXPRESSION_EVALUATION from $\mathcal{R}_0$ is applied if current expression corresponds to message expression, where receiver or some argument of this expression is not evaluated.

Let $\mathcal{R}_0^*(\mathcal{A}_0)$ be a computation

$$\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_i, \ldots$$

where for each $i > 0$ $\mathcal{A}_i = \mathcal{R}_0(\mathcal{A}_{i-1})$. Let $\mathcal{R}_1^*(\mathcal{B}_0)$ be a computation

$$\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_j, \ldots$$

where for each $j > 0$ $\mathcal{B}_j = \mathcal{R}_1(\mathcal{B}_{j-1})$. As shown in Section 6.4, we have that $\mathcal{R}_1^*(\mathcal{B}_0)$ can be shorter than $\mathcal{R}_0^*(\mathcal{A}_0)$.

Consider $\mathcal{A}_i$ and $\mathcal{B}_j$ such that $\mathcal{A}_i \equiv_\Pi \mathcal{B}_j$ and STATEMENT_EXECUTION from $\mathcal{R}_0$ and $\mathcal{R}_1$ applies to $\mathcal{A}_i$ and $\mathcal{B}_j$, respectively. This happens, if

Conditions 1

- $\mathcal{A}_i \equiv_\Pi \mathcal{B}_j$
- *the inequality*

$$\mathit{Expression\_value}\,(\mathit{Current\_expression}\,(\mathsf{S}),\mathsf{S}) \neq \mathsf{undef}$$

  holds in $\mathcal{A}_i$ and $\mathcal{B}_j$;
- *the equality*

$$\mathsf{Assignments} = \mathsf{empty}$$

  holds in $\mathcal{A}_i$ and $\mathcal{B}_j$;
- *Current_expression (S) is not the last expression of the method being evaluated.*

S denotes here the constant symbol Current_state. If Conditions 1 are satisfied, then the value of the term *Current_expression* (S) should be changed.

Conditions 1 imply, that if $\mathcal{R}_0$ is applied to $\mathcal{A}_i$, then the update rule

$$\mathit{Current\_expression}\,(\mathsf{S}) := f(\mathit{Current\_expression}\,(\mathsf{S}))$$

is used, where f is one of the function symbols: *Superexpression*, *Next_argument* or *Next_statement*. $\mathcal{R}_1$ applied to $\mathcal{B}_j$ uses the update rule

$$\mathit{Current\_expression}\,(\mathsf{S}) := \mathit{Cont}\,(\mathit{Current\_expression}\,(\mathsf{S}))$$

By definition of *Cont*, values of the term *Current_expression* (S) in $\mathcal{A}_{i+1}$ and $\mathcal{B}_{j+1}$ are not necessary equivalent. Hence the equivalence

$$\mathcal{A}_{i+1} \equiv_\Pi \mathcal{B}_{j+1}$$

does not hold because of the function symbol *Current_expression*. But the following equivalence holds

$$\mathcal{A}_{i+1} \equiv_\Phi \mathcal{B}_{j+1}$$

where $\Phi = \Pi/\{\mathit{Current\_expression}\}$.

Let e and e' belonging to EXPRESSIONS be values of the term *Current_expression* (S) in the algebras $\mathcal{A}_{i+1}$ and $\mathcal{B}_{j+1}$, respectively. Usually evaluation of e' will not start in $\mathcal{A}_{i+1}$. In such a case $\mathcal{R}_0$ is applied to $\mathcal{A}_{i+1}$ several times, until in some algebra $\mathcal{A}_k$ some e'' will be assigned to *Current_expression* (S), such that evaluation of e'' will begin in $\mathcal{A}_{k+1}$. *Cont* is defined in such a way, that evaluation of e' will begin in $\mathcal{B}_{j+1}$. Obviously, the conditions

$$\mathit{Expression\_value}\,(e'', \mathsf{S}) = \mathsf{undef}$$

and

$$\mathit{Expression\_value}\,(e', \mathsf{S}) = \mathsf{undef}$$

hold in $\mathcal{A}_k$ and $\mathcal{B}_{j+1}$, respectively. By Lemma 5.6 an order, in which expressions in given method are evaluated, is fixed. By Proposition 6.0.1 the function *Cont* follows this order. It implies, that e' and e'' are equivalent, hence

$$\mathcal{A}_k \equiv_\Pi \mathcal{B}_{j+1}$$

Let us summarize considerations given so far. Recall that $\mathcal{A}_0$ and $\mathcal{B}_0$ satisfy the assumption

$$\mathcal{A}_0 \equiv_\Pi \mathcal{B}_0$$

If $\mathcal{R}_0$ and $\mathcal{R}_1$ are applied to $\mathcal{A}_i$ and $\mathcal{B}_j$, respectively, and $\mathcal{A}_i \equiv_\Pi \mathcal{B}_j$, two cases are possible:

- the relation $\equiv_\Pi$ is preserved;
- the relation $\equiv_\Pi$ is not preserved, and there exists such $k > 0$, that $\mathcal{R}_0^k(\mathcal{A}_i)$ is equivalent to $\mathcal{B}_{j+1}$ up to Π.

So, the computation $\mathcal{R}_0^*(\mathcal{A}_0)$ can be divided into blocks such that each algebra $\mathcal{A}_k^i$ is equivalent to some $\mathcal{B}_k$ up to Φ, and the last $\mathcal{A}_k^{i_k}$ is equivalent to $\mathcal{B}_k$ up to Π.

$$\begin{array}{c|c|r|r|c} \mathcal{A}_0 & \dots & \mathcal{A}_j^1, \mathcal{A}_j^2, \dots, \mathcal{A}_j^{i_j} & \mathcal{A}_{j+1}^1, \mathcal{A}_{j+1}^2, \dots, \mathcal{A}_{j+1}^{i_{j+1}} & \dots \\ \mathcal{B}_0 & \dots & \mathcal{B}_j & \mathcal{B}_{j+1} & \dots \end{array}$$

It is clear that the computation $\mathcal{R}_0^*(\mathcal{A}_0)$ is finite if and only if the computation $\mathcal{R}_1^*(\mathcal{B}_0)$ is finite.

We summary all considerations in the theorem

Theorem 6.1 *Let $\mathcal{R}_0^*(\mathcal{A}_0)$ and $\mathcal{R}_1^*(\mathcal{B}_0)$ be finite. Let $\mathcal{R}_0^\infty(\mathcal{A}_0)$ be the last algebra of $\mathcal{R}_0^*(\mathcal{A}_0)$, let $\mathcal{R}_1^\infty(\mathcal{B}_0)$ be the last algebra of $\mathcal{R}_1^*(\mathcal{B}_0)$. Then*

$$\mathcal{R}_0^\infty(\mathcal{A}_0) \equiv_\Pi \mathcal{R}_1^\infty(\mathcal{B}_0)$$

We leave this theorem without any proof.

This theorem implies, that for a given program P, the last algebras $\mathcal{R}_0^\infty(\mathcal{A}_0)$ and $\mathcal{R}_1^\infty(\mathcal{B}_0)$ are equivalent, up to Π. It means, that for any program P the value of the constant Last_value, which denotes the result returned by P, has equivalent interpretation in $\mathcal{R}_0^\infty(\mathcal{A}_0)$ and $\mathcal{R}_1^\infty(\mathcal{B}_0)$.

Chapter 7

Smalltalk programs optimization

7.1 Introduction to optimization

Typically, target code produced from source code by a compiler is not the best code; it can often be improved to run faster or use less memory. Such improvement is done during a phase of compilation after parsing. The well–known machine–independent optimization techniques include constant folding, dead–code elimination, eliminating common subexpressions and various loop optimizations. The overview of these methods can be found in [ASU86].

In this chapter we consider two sources of runtime overhead and show how to eliminate it. The first source is specific for object–oriented languages with late binding, like Self ([US91]) or Smalltalk ([GR89]), where a name of method to be called is bound to method dynamically. It increases the computation time by a cost of traversing the class hierarchy each time a method is called.

The second source of runtime overhead concerns wider class of programming languages: any well–written program is divided into small pieces of code (subroutines, methods, etc.) which implies that a significant part of the entire computation includes invocations of subroutines and returns from them.

Most of the optimization techniques include a phase of code analysis which detects some patterns in program. For instance in order to detect common subexpression optimizer has to transform its input code into an acyclic graph. Then the graph is transformed back into a more efficient code.

The optimization methods we are going to discuss need program annotation by possible types (classes) of expressions. In Section 7.2 we give various methods of collecting such informations.

In Section 7.3.1 we transform our algebra into the form which can be used by a modified version of $\mathcal{R}_1$ to run Smalltalk program faster by statically binding a name of method with method. In Section 7.3.2 we give such a rule. Sections 7.4.1 and 7.4.3 present a technique called *inlining*, which can be remedy for the second problem.

7.2 Collecting informations

The methods of optimizing Smalltalk programs that we are going to discuss need a collection of receivers' classes, which are used as an additional input for optimization. We call a collection of receivers' classes a *class annotation*. In general, the problem to determine the class of a given expression for a given program and an input is undecidable.

Various methods of finding an approximation of the class annotation were proposed. Graver and Johnson designed a type system for Smalltalk ([GJ90]). Their solution needs manual type declaration of methods, and then types of expressions can be reconstructed. Palsberg and Schwartzbach in [PS91] give an algorithm for type reconstruction.

A completely different approach was suggested by Agesen and Hölzle in [AH95]. Their method, called *feedback*, extracts receivers' possible classes of a program P from a previous execution of P. The optimization of P is divided into two phases. In the first phase P is run and if some message expression of P is computed, then the class of the receiver of this expression is stored in memory. The second phase optimizes P using informations stored in the previous run of P. In [AH95] a comparison between type reconstruction and feedback is discussed.

In this chapter we define a new algebra $\mathcal{C}_0$ obtained from the algebra $\mathcal{B}_0$ by adding a new universe CLASS_LISTS, and three functions: *TypesOf*, *First* and *Method*. Elements of the universe CLASS_LISTS represent class annotations as lists of elements of the universe CLASSES. The first element of such a list is given by a function

$$First : \text{CLASS\_LISTS} \to \text{CLASSES}$$

We assume that for each element of the universe EXPRESSIONS a class annotation is given, e.g. as a result of one of methods mentioned above. For each $e \in \text{EXPRESSIONS}$ class annotation of e is given as the value of a function

$$TypesOf : \text{EXPRESSIONS} \to \text{CLASS\_LISTS}$$

If for $e \in \text{EXPRESSIONS}$ representing message expression a class of its receiver is determined uniquely, i.e. the list $TypesOf(Receiver(e))$ has only one element, then it is possible to determine, which method should be invoked if e is evaluated. For such e a function

$$Method : \text{EXPRESSIONS} \to \text{METHODS}$$

is defined. In Section 7.3.1 we give the definition of the value of *Method*.

In addition we define a transition rule $\mathcal{R}_2$ for $\mathcal{C}_0$, which is a modified version of the transition rule $\mathcal{R}_1$. Semantics defined by $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$ is equivalent to the semantics defined by $\langle \mathcal{C}_0, \mathcal{R}_2 \rangle$ in the sense that if for some Smalltalk program P $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$ defines a semantics of P, then $\langle \mathcal{C}_0, \mathcal{R}_2 \rangle$ defines an equivalent semantics of P, too.

7.3 Static dispatch

7.3.1 Methods transformation

In this section we modify the algebra $\mathcal{B}_0$ in order to use informations stored by the function *TypesOf*.

Suppose that for an element e of the universe EXPRESSIONS the length of the list *TypesOf* (e) is one, i.e. the class of the expression e is uniquely determined. Suppose also that e denotes a receiver of a method expression, i.e. there exists e' $\in$ EXPRESSIONS such that

$$Receiver\,(\mathsf{e'}) = \mathsf{e}$$

Then it is easy to determine which method will be called if computation comes to e': it is a value of the term *Method* (*Name* (e'), *First* (*TypesOf* (e))) or, if no method with the name *Name* (e') is defined in class *TypesOf* (e), then a suitable method is searched in class *Superclass* (*TypesOf* (e)), then in *Superclass* (*Superclass* (*TypesOf* (e))), etc. If a method is found in a class C, then we assign value of the term *Method* (*Name* (e'), C) to the term *Method* (e'). Otherwise, if no method is found, then the program is not correct.

Such a transformation performed for each element in the universe EXPRESSIONS makes transformed program run faster.

7.3.2 Program improvements

Now we are ready to define a rule $\mathcal{R}_2$ which uses functions defined in the algebra $\mathcal{C}_0$. The definition of $\mathcal{R}_2$ is obtained from the definition of $\mathcal{R}_1$ by some modifications.

First, we define a new rule MODIFIED_LOOKUP. In $\mathcal{R}_2$ the rule EXPRES-

---

Rule MODIFIED_LOOKUP
 if *Method* ($\underline{CurrExp}$) $\neq$ undef **then**
  Method := *Method* ($\underline{CurrExp}$)
  Receiver := *Expression_value* (*Receiver* ($\underline{CurrExp}$), Current_state)
 else
  METHOD_LOOKUP
 endif
EndRule

SION_EVALUATION differs from the rule EXPRESSION_EVALUATION from $\mathcal{R}_1$, since MODIFIED_LOOKUP is used instead of METHOD_LOOKUP.

Rule EXPRESSION_EVALUATION
if *is not method* **then**
PRIMARY_EVALUATION
elseif *Expression_value* (*Receiver* (*CurrExp*), Current_state) = undef **then**
CurrExp := *Receiver* (*CurrExp*)
elseif *First_argument* (*CurrExp*) = undef **then**
MODIFIED_LOOKUP
elseif *Expression_value* (*First_argument* (*CurrExp*), Current_state) = undef **then**
CurrExp := *First_argument* (*CurrExp*)
else
MODIFIED_LOOKUP
endif
EndRule

7.3.3 Equivalence of semantics

In this section we study equivalence of the continuation semantics given in Chapter 6, and its optimized version with static dispatch defined in Section 7.3.1. We are going to show, that the results given by both semantics are equivalent in the sense of the definitions of Section 6.5. All arguments given below are similar to the arguments given in Section 6.5 concerning computations $\mathcal{R}_0^*(\mathcal{A}_0)$ and $\mathcal{R}_1^*(\mathcal{B}_0)$.

Let P be a Smalltalk program. Let $\mathcal{B}_0$ be the algebra defined for P according to the definition given in Chapter 6, and let $\mathcal{C}_0$ be the algebra defined for P according to the definition given in Section 7.3.1. $\mathcal{R}_1$ and $\mathcal{R}_2$ are transition rules defined in Chapter 6 and Section 7.3.2, respectively. So, the pairs $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$ and $\langle \mathcal{C}_0, \mathcal{R}_2 \rangle$ define two operational semantics of P.

Recall Definition 6.1 of equivalence given in Section 6.5. Given a program P, two elements b and c belonging to $\mathcal{B}_0$ and $\mathcal{C}_0$, respectively, are equivalent

$$b \equiv c$$

if and only if b and c represent either the same name, either the same list of names, either the same object, either the same variable, either the same method, either the same expression, or the same state of P. Let Σ be any signature included in signatures $\Sigma_{\mathcal{B}}$ and $\Sigma_{\mathcal{C}}$ of $\mathcal{B}_0$ and $\mathcal{C}_0$, respectively. We shall write, that $\mathcal{B}_0$ and $\mathcal{C}_0$ are equivalent up to Σ

$$\mathcal{B}_0 \equiv_\Sigma \mathcal{C}_0$$

if for each term t built over Σ, interpretations of t in $\mathcal{B}_0$ and $\mathcal{C}_0$ are equivalent. Notice, that each set of applicable rules of $\mathcal{R}_1$ and $\mathcal{R}_2$ consists of at most one rule **extend**, hence definition of equivalence between elements can be extended to the newly created elements (see Definition 6.2 in Section 6.5).

The definition of $\mathcal{C}_0$ is derived from the definition of $\mathcal{B}_0$, by adding an additional function *Method*. So, if $\Sigma_{\mathcal{B}}$ denotes the signature of $\mathcal{B}_0$, then

$$\mathcal{B}_0 \equiv_{\Sigma_{\mathcal{B}}} \mathcal{C}_0$$

Let $\mathcal{R}_1^*(\mathcal{B}_0)$ be the computation

$$\mathcal{R}_1^*(\mathcal{B}_0) = \mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m, \ldots$$

determined by the pair $\langle \mathcal{B}_0, \mathcal{R}_1 \rangle$. Let $\mathcal{R}_2^*(\mathcal{C}_0)$ be the computation

$$\mathcal{R}_2^*(\mathcal{C}_0) = \mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n, \ldots$$

determined by the pair $\langle \mathcal{C}_0, \mathcal{R}_2 \rangle$.

Since $\mathcal{R}_1$ is different from $\mathcal{R}_2$, the computations $\mathcal{R}_1^*(\mathcal{B}_0)$ and $\mathcal{R}_2^*(\mathcal{C}_0)$ can be different. Below we consider these differences between $\mathcal{R}_1^*(\mathcal{B}_0)$ and $\mathcal{R}_2^*(\mathcal{C}_0)$.

Assume that $\mathcal{B}_i$ and $\mathcal{C}_j$ are such that

$$\mathcal{B}_i \equiv_{\Sigma_{\mathcal{B}}} \mathcal{C}_j$$

but

$$\mathcal{R}_1(\mathcal{B}_i) \equiv_{\Sigma_{\mathcal{B}}} \mathcal{R}_2(\mathcal{C}_j)$$

does not hold. It implies, that two different sets of rules have been applied, i.e. the sets of applicable rules $E_{\mathcal{B}_i}(\mathcal{R}_1)$ and $E_{\mathcal{C}_j}(\mathcal{R}_2)$ are different. Notice, there is only one difference between $\mathcal{R}_1$ and $\mathcal{R}_1$: in $\mathcal{R}_2$ LOOKUP_METHOD is replaced by MODIFIED_LOOKUP. It follow that if

$$\mathcal{B}_i \equiv_{\Sigma_{\mathcal{B}}} \mathcal{C}_j$$

and METHOD_LOOKUP is applicable to $\mathcal{B}_i$, and MODIFIED_LOOKUP is applicable to $\mathcal{C}_j$, then

- $\mathcal{B}_i \models Receiver\,(Current\_expression\,(\text{Current\_state})) \neq \text{undef}$
- $\mathcal{C}_j \models Receiver\,(Current\_expression\,(\text{Current\_state})) \neq \text{undef}$
- $\mathcal{C}_j \models Method\,(Current\_expression\,(\text{Current\_state})) \neq \text{undef}$

In such a case the values of the term *Current_expression* (Current_state) in $\mathcal{B}_i$ and $\mathcal{C}_j$ correspond to a message expression *exp* in the program *P*. Moreover, the method being called by *exp*, can be uniquely determined, and the element *m* corresponding to this method is the value of the term *Method* (*Current_expression* (Current_state)) in $\mathcal{C}_j$.

The rules METHOD_LOOKUP and MODIFIED_LOOKUP compute a value of the constant Method, which corresponds to the method being called. METHOD_LOOKUP (in $\mathcal{R}_1$) computes Method in several steps by traversing the class hierarchy. These steps form a subsequence of $\mathcal{R}_1^*(\mathcal{B}_0)$

$$\mathcal{B}_i, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_{i_k}$$

For each n, $i \leq n < i_k$ we have

$$\mathcal{B}_n \models \text{Method} = \text{undef}$$

and

$$\mathcal{B}_{i_k} \models \mathsf{Method} \neq \mathsf{undef}$$

Notice, that METHOD_LOOKUP applied to $\mathcal{B}_n$, $i \leq n < i_k$, changes only the value of the constant Class. It implies, that for each n such that $i \leq n < i_k$

$$\mathcal{B}_n \equiv_{\Theta} \mathcal{C}_j$$

where $\Theta = \Sigma_{\mathcal{B}}/\{\mathsf{Class}\}$.

In the final stage $\mathcal{B}_{i_k}$ a value is assigned to Method by METHOD_LOOKUP. In $\mathcal{R}_2(\mathcal{C}_0)$ a value of Method is determined in one step, and is already computed in $\mathcal{R}_2(\mathcal{C}_j)$ by evaluating the term *Method* (*Current_expression* (Current_state)), and assigning its value to Method. If the function *Method* is determined correctly (see Section 7.2), then values of the constant Method in $\mathcal{B}_{i_k}$ and $\mathcal{R}_2(\mathcal{C}_j)$ should be equivalent, hence

$$\mathcal{B}_{i_k} \equiv_{\Sigma_{\mathcal{B}}} \mathcal{R}_2(\mathcal{C}_j)$$

Recall that

$$\mathcal{B}_0 \equiv_{\Sigma_{\mathcal{B}}} \mathcal{C}_0,$$

so two computations: $\mathcal{R}_1^*(\mathcal{B}_0)$ and $\mathcal{R}_2^*(\mathcal{C}_0)$, begin from two equivalent algebras. Moreover, by Proposition 6.0.3 if

$$\mathcal{B}_i \equiv_{\Sigma_{\mathcal{B}}} \mathcal{C}_j$$

then if sets of applicable rules $E_{\mathcal{B}_i}(\mathcal{R}_1)$ and $E_{\mathcal{C}_j}(\mathcal{R}_2)$ are equal, then

$$\mathcal{R}_1(\mathcal{B}_i) \equiv_{\Sigma_{\mathcal{B}}} \mathcal{R}_2(\mathcal{C}_j)$$

Notice, that if $E_{\mathcal{B}_i}(\mathcal{R}_1)$ and $E_{\mathcal{C}_j}(\mathcal{R}_2)$ are equal, then all terms in $E_{\mathcal{B}_i}(\mathcal{R}_1)$ and $E_{\mathcal{C}_j}(\mathcal{R}_2)$ are built over $\Sigma_{\mathcal{B}}$. If these sets of applicable rules are not equal, then $\mathcal{R}_1(\mathcal{B}_i)$ and $\mathcal{R}_2(\mathcal{C}_j)$ are not equivalent up to $\Sigma_{\mathcal{B}}$. However, in such a case there exists nonnegative k such that

$$\mathcal{R}_1^{(k)}(\mathcal{B}_i) \equiv_{\Sigma_{\mathcal{B}}} \mathcal{R}_2(\mathcal{C}_j)$$

So, the computations $\mathcal{R}_1^*(\mathcal{B}_0)$ and $\mathcal{R}_2^*(\mathcal{C}_0)$ become equivalent again from $\mathcal{R}_1^k(\mathcal{B}_i)$ and $\mathcal{R}_2(\mathcal{C}_j)$. Hence, we have established two theorems.

Theorem 7.1 *The computation $\mathcal{R}_1^*(\mathcal{B}_0)$ is finite if and only if the computation $\mathcal{R}_2^*(\mathcal{C}_0)$ is finite.*

Theorem 7.2 *Let computations $\mathcal{R}_1^*(\mathcal{B}_0)$ and $\mathcal{R}_2^*(\mathcal{C}_0)$ be finite. Let $\mathcal{R}_1^\infty(\mathcal{B}_0)$ be the last algebra of $\mathcal{R}_1^*(\mathcal{B}_0)$, let $\mathcal{R}_2^\infty(\mathcal{C}_0)$ be the last algebra of $\mathcal{R}_2^*(\mathcal{C}_0)$. Then*

$$\mathcal{R}_1^\infty(\mathcal{B}_0) \equiv_{\Sigma_{\mathcal{B}}} \mathcal{R}_2^\infty(\mathcal{C}_0)$$

Theorem 7.2 implies, that the final value of the constant Last_value, which represents the value of the program P, has equivalent interpretations in $\mathcal{R}_1^\infty(\mathcal{B}_0)$ and $\mathcal{R}_2^\infty(\mathcal{B}_0)$.

7.4 Inlining

7.4.1 Introduction to inlining

Inlining technique is used if it is known in compiling time that a method M_1 will call a method M_2. Then inlining consist of inclusion of the body of the method M_2, which replaces calling of M_2. In C++ ([Str91]) method inlining is a part of language definition. Comments on method inlining and speed-ups in SELF may be found in [HCU91].

In this section we discuss inlining in Smalltalk. We give transformation of an abstract form of a program instead of transformation of a source program, to be able to study equivalence of an original and the transformed program.

Example

Now we give an example of inlining. We consider the projections of points of three dimensional space onto two dimension surface. Suppose that points are represented in Smalltalk as objects of class `ThreeDim`, and methods `x`, `y` and `x` are defined for these objects, which return x–th, y–th and z–th coordinates, respectively.

Suppose that a class ***Projections*** represents a projection and methods `xcoord:xcoord:xcoord:` and `ycoord:ycoord:ycoord:` are defined, which transforms three dimensions coordinates into coordinates of a plane. Consider the following of code:

Example 7.1

```
sack ← Bag new.
sack add: (ThreeDim new).
sack add: (ThreeDim new).
sack add: (ThreeDim new).
...
sack add: (ThreeDim new).
mypen ← Pen new.
sack do: [ :point |
    x ← (Projections
           xcoord: (point x) xcoord: (point y) xcoord: (point z)).
    y ← (Projections
           ycoord: (point x) ycoord: (point y) ycoord: (point z)).
    mypen goto: x@y].
```

The variable `sack` represents a collection of objects, in this particular case a set of three dimensional points. The statement `'sack do: [:point | ...]'` is performed as follows: for each element `e` in `sack` `e` is assigned to `point` and statements followed by | mark are performed. First new coordinates are computed and then a line to the point `x@y` is drawn. If the method `xcoord:xcoord:xcoord:` is given as

```
xcoord:  x xcoord:  y xcoord:  z
↑(10 - x) + (20 - y)*0.5*(-0.5)
```

and `ycoord:ycoord:ycoord:` as[1]

```
ycoord:  x ycoord:  y coord:  z
↑(30 - z) + (20 - y)*0.5*(0.8660254)
```

where (10,30,20) is the point of view of an observer, we can replace the fragment of Example 7.1

```
x ← (Projections
  xcoord:  (point x) xcoord:  (point y) xcoord:  (point z)).
y ← (Projections
  ycoord:  (point x) ycoord:  (point y) ycoord:  (point z)).
```

by the following statements:

```
x ← (10 - (point x)) + (20 - (point y))*0.5*(-0.5).
y ← (30 - (point z)) + (20 - (point y))*0.5*(0.8660254).
```

It is easy to notice that such a conversion reduces the number of callings. Overhead of calling and returning is different in different languages, but saving few instruction in loop may dramatically decrease the computation time.

Notes on inlining

Inlining is possible only in special cases, which we shall discuss in Section 7.4.2. In order to perform inlining transformation of some method, we have to know which code could be inlined. It requires knowledge of receiver's class, which provides information which method will be called during future computation. These techniques are studied in Section 7.2.

7.4.2 Limitations of inlining

The technique of inlining can be applied only in a limited number of cases. In this section we shall discuss all of them.

It is easy to notice, that presence of object, class, and shared variables in a method to be inlined causes trouble. Consider two methods: *m* and *n*, where the body of *n* can be inlined in place of a message expression *exp* in *m*. In the general case if *n* is defined in a class different from the class of *m*, and *n* refers to some class or shared variable V, then usually the variable name V in *m* refers to a different variable. Moreover, it is possible, that in scope of *m* no variable named V is declared. So, if *n* refers to a class or shared variable, then inlining is possible only if a method class of *n* is a superclass of a class of *m*.

[1]It is so called *cavalier projection* (see [FvDFH90]).

Another limitation concerns object variables. Suppose that *m* and *n* are defined in the same class, and *n* refers to some object variable v. For each object of a given class a new set of object variables is created. Therefore, except of special cases, names of object variables used in *m* and *n* refer to different variables. So, in the general case only those methods *n*, which do not refer to names of object variables, can be inlined into another method.

Additionally, a method, which includes a block expression with return statement, can not be inlined into another method. Let us consider a result of inlining such a method. Let *P* be a program containing a class ***Test***. Let *m* and *n* be two methods declared in ***Test*** as follows:

```
m
(self n) value
```

```
n
↑[↑2]
```

Then the expression

```
(Test new) m
```

raises an error, because evaluation of the block object `'[↑2]'` tries to finish the computation in the state where `'[↑2]'` has been created, but the computation in this state has already been finished (see Section 5.6.1). Since an object being the value of *self* (in *m*) belongs to the class ***Test***, we can replace *'self n'* by the body of *n* obtaining a new method *mnew*

```
mnew
[↑2] value
```

which does not raise any error. Therefore we do not consider inlining of methods consisting of blocks with return statements.

The last limitation of inlining concerns recursive methods. In this case, applying inlining to a recursive method causes that this method appears again.

7.4.3 Method conversion

The inlining conversion presented here is split into two phases. In the first phase the method being inlined is converted into an equivalent form, which is suitable for substitution; we perform the so–called α–conversion, and some others changes. Given a program *P*, and its algebraic representation $\mathcal{C}_0$ defined in Section 7.2, we shall represent the transformed program by an algebra $\tilde{\mathcal{C}}_0$ of the same signature as $\mathcal{C}_0$.

In the second phase given in Section 7.4.4, we inline the method so transformed into another method. By this transformation we obtain another algebra $\mathcal{D}_0$ with the same signature as the algebra $\tilde{\mathcal{C}}_0$. The transition rule $\mathcal{R}_2$ defined for $\mathcal{C}_0$ is also a correct transition rule for $\tilde{\mathcal{C}}_0$ and $\mathcal{D}_0$, and we do not need to modify $\mathcal{R}_2$.

We shall use the following symbols: $\mathcal{M} \in$ METHODS represents a method *M*, $\mathcal{N} \in$ METHODS represents a method *N*, and a subexpression *exp* of *M* is represented by $e_{\mathcal{N}} \in$ EXPRESSIONS. We assume that it is possible to inline *N* into *M* in place of *exp*, so the value of the term $\mathit{Method}(e_{\mathcal{N}})$ is defined, and is equal to $\mathcal{N}$.

7.4.3.1 Conversion of temporary variable names

In order to inline the method N in the method M we have to rename all names of temporary variables of N to make sure, that the sets of temporary names in M and in N are disjoint. This operation is similar to the transformation called α–conversion in the lambda calculus. As defined in Section 4.2.3, names of temporary variables of $\mathcal{N}$ are stored by the value of the function *Temporary_names* on $\mathcal{N}$. To avoid conflicts we replace the list $Temporary\_names\,(\mathcal{N}) = [var_1 \ldots var_n]$ by $Temporary\_names\,(\mathcal{N}) = [var'_1 \ldots var'_n]$, where the variable names var'_i are not used in $\mathcal{M}$. Then for each var_i we replace all its occurrences in $\mathcal{N}$ by var'_i. Notice, that var_i may occur in $\mathcal{N}$ as an element of the list $Assignments\,(e)$, where e is an expression of $\mathcal{N}$, and belongs to the universe EXPRESSIONS. Moreover, var_i may be a value of the term $Name\,(e)$, where e is not message expression, and finally var_i may occur in the list *Argument_names* of argument names of block.

This conversion does not change the meaning of the method $\mathcal{N}$, since it changes only names of local variables.

The program so modified will be represented by an algebra $\tilde{\mathcal{C}}_0^1$, a modified version of $\mathcal{C}_0$.

7.4.3.2 Conversion of argument names

In order to avoid conflicts of names of arguments of the method N and names of temporary variables and arguments of the method M, we convert argument names of N in the same way as we have changed temporary names in Section 7.4.3.1. Remind, that arguments of N are represented by a list of names being the value of the term $Argument\_names\,(\mathcal{N})$. The conversion of argument names also does not change the meaning of the method M.

The program so modified will be represented by an algebra $\tilde{\mathcal{C}}_0^2$, a modified version of $\tilde{\mathcal{C}}_0^1$.

7.4.3.3 Conversion of global and class variables

It is not necessary to make any conversion of global and class variables, since names of global variables have the same interpretation in M and in N. Recall, that M and N do not include pool and object variables (see Section 7.4.2).

7.4.3.4 *return* statement

Every method in Smalltalk returns a value. In some cases it is explicitly stated which value should be returned by marking some statement by an arrow ↑. In such a case the value of expression on the right side of ↑ is returned. Otherwise, the value of the receiver of the method is returned.

For a reason which will be clear later, if N does not terminate by a statement with the ↑ symbol, then at the end of N we append a new statement

```
↑self
```

The program transformed in this way will be represented by the algebra $\tilde{\mathcal{C}}_0$ being a modification of $\tilde{\mathcal{C}}_0^2$. In $\tilde{\mathcal{C}}_0$ we extend the set EXPRESSIONS by a new element $e_\uparrow$ which has the following properties:

- $Return\_prefix\,(e_\uparrow) = \uparrow$
- $Name\,(e_\uparrow) =$ self
- if $e_{pre\uparrow}$ is the last expression of $\mathcal{N}$ in $\tilde{\mathcal{C}}_0^2$, then in $\tilde{\mathcal{C}}_0$ we put

$$Cont\,(e_{pre\uparrow}) = e_\uparrow.$$

Clearly $\mathcal{N}$ outputs the same result as before.

7.4.4 Method inline

All modifications of the method N and its representation $\mathcal{N}$ given in Section 7.4.3 are needed to prepare for the phase of inlining. Inlining consists of three steps: extension of *Temporary_names* of $\mathcal{M}$, modifications concerning arguments of N, extension of EXPRESSIONS, and modification of the function *Cont*. We describe these operations below. As a result we obtain an algebra $\mathcal{D}_0$.

7.4.4.1 Temporary names

All temporary names on the list $Temporary\_names\,(\mathcal{N})$ are added to the list $Temporary\_names\,(\mathcal{M})$.

7.4.4.2 Modifications concerning argument names of N

If N has arguments, then in all message expressions of the program P where N is called, some expressions are referred to as parameters of N. These message expressions are of the form

receiver s1: p1 s2: p2 ... sn: pn

where *'s1:s2:...sn:'* is the signature matching the signature of N, and $p1$, $p2$, ..., pn are parameters of N to be called. During the computation parameters pi's are evaluated, and passed to N. Strictly speaking, first all parameters are evaluated in turn. Next, values of these parameters are assigned to arguments of N, and finally in N these values are used. So, in order to preserve the meaning of P when N is inlined into M, we have to modify representation of M, so that after inlining of N into M the value of each parameter pi will be assigned to the corresponding argument of N.

In order to satisfy these requirements, we perform the following modifications. Let the list $[a_1, a_2, \ldots, a_n]$ of argument names be the value of $Argument\_names\,(\mathcal{N})$, where each a_i represents i-th argument of N, $\mathcal{M}$ represents M, and $\mathcal{N}$ represents N. First, each a_i is added to the list $Temporary\_names\,(\mathcal{M})$. By Section 7.4.3.2 lists $Argument\_names\,(\mathcal{N})$ and

Temporary_names $(\mathcal{M})$ are disjoint. Next, each a_i is added to the list *Assignments* (e_i), where $e_i \in$ EXPRESSIONS is the representation of *pi*. Hence, during the computation the value of the parameter *pi* is assigned to the argument name *ai*.

Recall, that if the point of inlining is represented by $e_{\mathcal{N}}$, then e_i is the value of the term

$$Next\_argument^{(i-1)}(First\_argument(e_{\mathcal{N}})).$$

7.4.4.3 Extension of the set EXPRESSIONS

The operation of inlining of N into M copies all expressions of N into M, but the original method N remains in the program P. Therefore we duplicate elements of EXPRESSIONS representing expressions of N. The duplicates represent the copy of N inlined into M. So, for each element $e \in$ EXPRESSIONS representing an expression of N, we add to EXPRESSIONS a copy $\overline{e}$ of e. The subset of EXPRESSIONS consisting of these copies is denoted by $Copy_{\mathcal{N}}$.

To define $\mathcal{D}_0$ we have to define functions of $\tilde{\mathcal{C}}_0$ for arguments belonging to $Copy_{\mathcal{N}}$. If f is one of the function symbols

$$Name\,,\ Argument\_names\,,\ Constant\_value\,,\ Assignments\,,\ Return\_prefix$$

defined on EXPRESSIONS, then for each $\overline{e} \in Copy_{\mathcal{N}}$ we put

$$f(\overline{e}) = f(e) \tag{7.1}$$

where $\overline{e}$ is the copy of e.

By $\mathcal{F}$ we denote the set

$$\{$$
$$First\_statement\,,\ Receiver\,,\ First\_argument\,,\ Cont\,,$$
$$Next\_argument\,,\ Superexpression\,,\ Next\_statement$$
$$\}$$

of function symbols defined on EXPRESSIONS and having values in EXPRESSIONS. For each $f \in \mathcal{F}$ we extend the definition of f as follows: for each $\overline{e} \in Copy_{\mathcal{N}}$ we put

$$f(\overline{e}) = \overline{e}' \tag{7.2}$$

where $\overline{e}$ and $\overline{e}'$ are copies of e and e', respectively, and $f(e) = e'$.

Notice that the extension (7.1) of functions preserves "type" of expression. For instance, if $e \in$ EXPRESSIONS represents some constant, say 4, i.e. *Constant_value* $(e) = 4$, then by the definition of extension *Constant_value* $(\overline{e}) = 4$, i.e. $\overline{e}$ also represents the constant 4.

The extension (7.2) preserves relations between expressions. For instance, suppose that the receiver of the expression *exp* is *exp*$'$, and e and e' represent *exp* and *exp*$'$, respectively. The relationship between e and e' is denoted by *Receiver* $(e) = e'$. So, if $\overline{expr}$ and $\overline{expr}'$ are copies of expressions *expr* and *expr*$'$, respectively, $\overline{e}$ denotes $\overline{expr}$, and $\overline{e}'$ denotes $\overline{expr}'$, then *Receiver* $(\overline{e}) = \overline{e}'$.

7.4.4.4 Modification of the function *Cont*

Let *expr* be the expression, which is replaced by the body of the method N. Now, we are going to modify *Cont* defined in $\tilde{C}_0$ in order to point to inlined body of N instead of *expr*.

If before modification of M, the next expression to be evaluated after *expr'* is *expr*, then in the modified method M the first expression of N will be evaluated after expression *expr'*. By $e_{\mathcal{N}} \in$ EXPRESSIONS we denote the representation of *expr*.

For each $e \in$ EXPRESSIONS such that in $\tilde{C}_0$

$$Cont\,(e) = e_{\mathcal{N}}$$

we put

$$Cont\,(e) = \overline{f}$$

where $\overline{f} \in \mathcal{D}_0$ is the copy of the value of the term $First\_expression\,(\mathcal{N})$.

We are ready now to show how to continue computation after execution of the inlined code of $\mathcal{N}$. In Section 7.4.3.4 we have performed a conversion which guarantee that N has return statement. It is not possible to deal with the returned value directly. We solve this problem as follows: we introduce a new temporary variable with a fresh name *'var'*. Then we extend the list $Temporary\_names\,(\mathcal{M})$ by *'var'* and we put

$$Name\,(e_{\mathcal{N}}) = {'var'}$$

$$Receiver\,(e_{\mathcal{N}}) = \mathsf{undef}$$

$$First\_argument\,(e_{\mathcal{N}}) = \mathsf{undef}.$$

Now, once again we shall modify some functions for arguments belonging to $Copy_{\mathcal{N}}$. Let $f \in$ EXPRESSIONS be the last expression of $\mathcal{N}$. We have $Cont\,(\mathrm{f}) = \mathsf{undef}$. We make the following modifications:

- the list $Assignments\,(\overline{f})$ is extended by *'var'*;
- we put

$$Cont\,(\overline{f}) = e_{\mathcal{N}}$$

$$Return\_prefix\,(\overline{f}) = \mathsf{undef}$$

where $\overline{f}$ is the copy of f.

If the pseudovariable *self* or *super* occurs in $\mathcal{N}$, then

- a new temporary variable, *'rec_var'*, is added to the method $\mathcal{M}$, i.e. the list $Temporary\_variable\,(\mathcal{M})$ is extended by *'rec_var'*;
- each occurrence of *self* is replaced by *'rec_var'*, i.e. we put

$$Name\,(\overline{f}) = {'rec\_var'}$$

  for each $f \in$ EXPRESSIONS such that f corresponds to some expressions of N and the equality $Name\,(f)$ = *'self'* holds in $\tilde{C}_0^2$;

- *'rec_var'* is added to the list $Assignments\,(\overline{r})$, where $r = Receiver\,(e_{\mathcal{N}})$, and $\overline{r}$ is the copy of r. So, if $\overline{r}$ is evaluated, then its value, which corresponds to the value of *self* in N, is assigned to *'rec_var'*.

An additional problem related to *super* is discussed in Section 7.4.4.5.

7.4.4.5 *super* conversion

Inlining of a method which includes *super* leads to two problems. The first is how to compute the value of *super*, the second is related to the fact, that *super* modifies method lookup, when *super* is the receiver of a message expression. The solution of the first problem is identical to the solution of the problem related to *self* presented in Section 7.4.4.4; i.e. consists of replacing each occurrence of *super* by the variable name *'rec_var'*. For each $f \in$ EXPRESSIONS such that f corresponds to some expressions of N, and the equality $Name\,(f) = {}'self'$ holds, we put

$$Name\,(\overline{f}) = {}'rec\_var'$$

where $\overline{f}$ is the copy of f.

A solution of the second problem is also easy. We use an observation, that if *super* is the receiver of a message expression s, then since we know the class of the method $\mathcal{N}$ and the signature of this expression, the method to be called in s can be easily fixed before computation. If this is a method P represented by $\mathsf{p} \in$ METHODS, then we put

$$Method\,(\overline{s}) = \mathsf{p}$$

where $\overline{s}$ is the copy of s.

7.5 Equivalence between semantics

Given a program P, we have defined five representations of P: $\mathcal{C}_0$, $\tilde{\mathcal{C}}_0^1$, $\tilde{\mathcal{C}}_0^2$, $\tilde{\mathcal{C}}_0$, and $\mathcal{D}_0$. $\mathcal{C}_0$ is a representation of P with the continuation function *Cont*. $\tilde{\mathcal{C}}_0^1$ is derived from $\mathcal{C}_0$ by changing names of temporaries, and $\tilde{\mathcal{C}}_0^2$ is derived from $\tilde{\mathcal{C}}_0^1$ by changing names of arguments of a method N. By adding *return* statement at the end of N the algebra $\tilde{\mathcal{C}}_0$ has been obtained from $\tilde{\mathcal{C}}_0^2$. $\mathcal{D}_0$ is derived from $\tilde{\mathcal{C}}_0$ by inlining N into method M.

In this section we sketch proofs that $\mathcal{C}_0$, $\tilde{\mathcal{C}}_0^1$, $\tilde{\mathcal{C}}_0^2$, $\tilde{\mathcal{C}}_0$, and $\mathcal{D}_0$ are equivalent.

7.5.1 Equivalence between $\mathcal{C}_0$ and $\tilde{\mathcal{C}}_0$

Assume that an ASM $\langle \mathcal{C}_0, \mathcal{R}_2 \rangle$ gives the semantics of P as defined in Section 7.3.2, and ASM $\langle \mathcal{D}_0, \mathcal{R}_2 \rangle$ gives the semantics of P, as defined in Section 7.4.3. The notion of equivalence is given by the definition of Section 7.3.3.

We begin with two theorems.

Theorem 7.3 *Let $\Delta = \Sigma \backslash \{Name, Assignments, Temporary\_names\}$, where Σ denotes the signature of $\mathcal{C}_0$. Then for each nonnegative i*

$$\mathcal{C}_i \equiv_\Delta \tilde{\mathcal{C}}_i^1$$

holds.

Theorem 7.4 *Let $\Delta = \Sigma \backslash \{Name, Argument\_names\}$, where Σ denotes the signature of $\mathcal{C}_0$. Then for each nonnegative i*

$$\tilde{\mathcal{C}}_i^1 \equiv_\Delta \tilde{\mathcal{C}}_i^2$$

holds.

The proofs of these theorems are very similar to the proof of Theorem 6.1, so we skip them.

The equivalence between $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$ is not as immediate as in theorems presented above, since $\tilde{\mathcal{C}}_0$ is derived from $\tilde{\mathcal{C}}_0^2$ by adding some elements and extending definitions of some functions. Let us analyze differences between these algebras.

Recall the definition (Definition 6.1) of equivalence of elements of two algebras, say $\mathcal{A}$ and $\mathcal{B}$, defined for a program P. Two elements $a \in \mathcal{A}$ and $b \in \mathcal{B}$ are equivalent

$$a \equiv b$$

if a and b represent the same element of P, e.g. the same method, object, or class. In the same way as in Definition 6.1, we define equivalence of elements of $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$. Notice, that there is no element in $\tilde{\mathcal{C}}_0^2$ equivalent to $e_\uparrow \in \tilde{\mathcal{C}}_0$, where $e_\uparrow$ is defined in Section 7.4.3.4.

Unfortunately, the property that for each function symbol f, and for each $e \in \tilde{\mathcal{C}}_0^2$, $e' \in \tilde{\mathcal{C}}_0$:

$$e \equiv e' \Rightarrow f_c(e) \equiv f_d(e')$$

does not hold, where f_c and f_d are functions in $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$ respectively, representing to f. The counterexample is provided by the function *Cont*. Therefore, the definition of equivalence of two algebras presented in Section 6.5 can not be applied to $\tilde{\mathcal{C}}_0$ and $\mathcal{C}_0$. Therefore we are going to give another definition of equivalence.

In an algebra $\mathcal{A}$ the definition of any (e.g. unary) function f can be given by the set of equalities

$$\{f(e_i) = t_i \mid e_i \in \mathcal{A}, t_i \in \mathcal{A}, i \in I\}$$

where I is a set of indices. We shall define a notion of equivalence using this description of a function.

We define equivalence of two algebras $\mathcal{A}$ and $\mathcal{B}$ as follows. By Σ we denote the signature of $\mathcal{A}$ and $\mathcal{B}$. By Σ_i we denote the subset of Σ containing all i–ary function symbols of Σ. Assume that all symbols in Σ have the arity less or equal to k. Let U be the set of elements of $\mathcal{A}$. By $S(\mathcal{A})$ we denote the set

$$\bigcup_{i=0}^{k} (\Sigma_i \times \underbrace{U \times U \times \ldots \times U}_{i \text{ times}}). \tag{7.3}$$

Given the relation $\equiv$ of equivalence among elements of $\mathcal{A}$ and $\mathcal{B}$, we define equivalence of two algebras $\mathcal{A}$ and $\mathcal{B}$ as follows.

Definition 7.1 *An algebra $\mathcal{B}$ is equivalent to algebra $\mathcal{A}$ up to a set $T \subseteq S(\mathcal{A})$ ($\mathcal{A} \equiv_T \mathcal{B}$ in symbols) if and only if for each k–ary function symbol f, and for each $a_1, a_2 \ldots a_k \in \mathcal{A}$ such that*

$$(f, a_1, a_2, \ldots, a_k) \in T$$

there exist $b_1, b_2, \ldots, b_k \in \mathcal{B}$ such that

$$a_1 \equiv b_1, a_2 \equiv b_2, \ldots, a_k \equiv b_k$$

and

$$f_{\mathcal{A}}(a_1, a_2, \ldots, a_k) \equiv f_{\mathcal{B}}(b_1, b_2, \ldots, b_k)$$

where $f_{\mathcal{A}}$ and $f_{\mathcal{B}}$ are functions in $\mathcal{A}$ and $\mathcal{B}$, respectively, corresponding to f.

Notice, that the relation $\equiv$ between structures is not symmetric, $\mathcal{B}$ can contain more elements and functions than $\mathcal{A}$.

Let us study, how application of the update rule R

$$f(t_1, t_2, \ldots, t_n) = t$$

to equivalent algebras $\mathcal{A}$ and $\mathcal{B}$ preserves equivalence of the results of the application. R changes the sets of equalities defining functions in $\mathcal{A}$ and $\mathcal{B}$. But if certain conditions are satisfied, then both sets of equalities are changed in equivalent way. We give these conditions in the following lemma:

Lemma 7.1 *Let T be a set such that if*

$$\mathcal{A} \equiv_T \mathcal{B}.$$

Let R be the rule

$$f(t_1, t_2, \ldots, t_n) = t.$$

If there exists

$$(f, a_1, a_2, \ldots, a_n) \in T$$

such that

- *for each i, the value of t_i in $\mathcal{A}$ is a_i, and*
- *values of the term t in $\mathcal{A}$ and $\mathcal{B}$ are equivalent,*

then $R(\mathcal{A})$ and $R(\mathcal{B})$ are equivalent up to T

$$R(\mathcal{A}) \equiv_T R(\mathcal{B})$$

We leave this lemma without any proof. Notice, that this lemma can be extended to the case when R is the set of several non clashing update rules satisfying given conditions.

A more complicated analysis is required, if R contains a rule **extend**. This rule, applied to $\mathcal{A}$ and $\mathcal{B}$, introduces a new element e to $\mathcal{A}$, and a new element e' to $\mathcal{B}$. We declare that e and e' are equivalent. Additionally, $\mathcal{A}$ and $\mathcal{B}$ are extended by some equalities defining some functions on e and e', respectively. If all terms in R have equivalent values in $\mathcal{A}$ and $\mathcal{B}$, then we define the extension $R(T)$ of T as follows: $R(T)$ is the least set such that

- $T \subseteq R(T)$
- $(f, a_1, a_2, \ldots, a_n) \in R(T)$ if there exists an update rule

$$f(t_1, t_2, \ldots, t_n) = t_{n+1}$$

of R, where for each i: a_i is the value of t_i in $\mathcal{A}$.

So, if conditions given in Lemma 7.1 are satisfied, and R consists of one **extend** rule, then

$$R(\mathcal{A}) \equiv_{R(T)} R(\mathcal{B}). \tag{7.4}$$

If R does not contain **extend** rule, then we put $R(T) = T$.

Now we are going to apply the notion of equivalence introduced above to algebras $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$. By Section 7.4.3.4 $\tilde{\mathcal{C}}_0$ is derived from $\tilde{\mathcal{C}}_0^2$ by the following changes:

- the element $e_{\uparrow}$ is added to EXPRESSIONS in $\tilde{\mathcal{C}}_0$;
- the definition of the function *Cont* is changed;
- the functions *Return_prefix* and *Name* are defined on $e_{\uparrow}$.

So, in $\tilde{\mathcal{C}}_0^2$ we have:

$$Cont\,(e'_{pre\uparrow}) = \mathsf{undef}$$

and in $\tilde{\mathcal{C}}_0$ we have

$$\begin{array}{l} Cont\,(e_{pre\uparrow}) = e_{\uparrow} \\ Return\,(e_{\uparrow}) = {'\uparrow'} \\ Name\,(e_{\uparrow}) = {'self'} \\ Cont\,(e_{\uparrow}) = \mathsf{undef} \end{array}$$

where $e'_{pre\uparrow}$ and $e_{pre\uparrow}$ are equivalent.

Let $S(\tilde{\mathcal{C}}_0^2)$ be the set

$$(\Sigma_0) \cup (\Sigma_1 \times U(\tilde{\mathcal{C}}_0^2)) \cup (\Sigma_2 \times U(\tilde{\mathcal{C}}_0^2) \times U(\tilde{\mathcal{C}}_0^2))$$

where Σ_k denotes the set of k–ary function symbols of the signature of $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$, and $U(\tilde{\mathcal{C}}_0^2)$ denotes the set of elements of $\tilde{\mathcal{C}}_0^2$. Notice, that signatures of $\tilde{\mathcal{C}}_0^2$ and $\tilde{\mathcal{C}}_0$ are equal, and the highest arity of function symbols is 2. We define

$$T = S(\tilde{\mathcal{C}}_0^2) \setminus \{(Cont\,, e'_{pre\uparrow})\}. \tag{7.5}$$

By the construction of $\tilde{\mathcal{C}}_0$ given in Section 7.4.3.4 we have

$$\tilde{\mathcal{C}}_0^2 \equiv_T \tilde{\mathcal{C}}_0.$$

Consider two computations:

$$\mathcal{R}_2^*(\tilde{C}_0^2) = \tilde{C}_0^2, \tilde{C}_1^2, \tilde{C}_2^2, \dots, \tilde{C}_m^2, \dots$$

and

$$\mathcal{R}_2^*(\tilde{C}_0) = \tilde{C}_0, \tilde{C}_1, \tilde{C}_2, \dots, \tilde{C}_n, \dots.$$

By (7.4) initial fragments of $\mathcal{R}_2^*(\tilde{C}_0^2)$ and $\mathcal{R}_2^*(\tilde{C}_0)$ are equivalent, i.e. there exists n such that for each k, $0 \leq k < n$

$$\mathcal{R}_2^{(k)}(\tilde{C}_0^2) \equiv_{\mathcal{R}_2^{(k)}(T)} \mathcal{R}_2^{(k)}(\tilde{C}_0).$$

But $\tilde{C}_0$ and $\mathcal{D}_0$ are not isomorphic, so it is possible that for some n

$$\mathcal{A} = \mathcal{R}_2^{(n)}(\tilde{C}_0^2) \text{ and } \mathcal{B} = \mathcal{R}_2^{(n)}(\tilde{C}_0)$$

and

$$\mathcal{A} \equiv_{\mathcal{R}_2^{(n)}(T)} \mathcal{B}$$

but

$$\mathcal{R}_2(\mathcal{A}) \not\equiv_{\mathcal{R}_2^{(n+1)}(T)} \mathcal{R}_2(\mathcal{B}).$$

It happens, if at least one of the conditions holds:

(1) the sets of applicable rules $E_{\mathcal{A}}(\mathcal{R}_2)$ and $E_{\mathcal{B}}(\mathcal{R}_2)$ are not equal, i.e. at least one condition in $\mathcal{R}_2$ evaluates to different values in $\mathcal{A}$ and $\mathcal{B}$;

(2) some term t in $E_{\mathcal{A}}(\mathcal{R}_2) \cap E_{\mathcal{B}}(\mathcal{R}_2)$ evaluates to different values in $\mathcal{A}$ and $\mathcal{B}$.

In (7.5) the set T has been obtained from $S(\tilde{C}_0^2)$ by removing $(Cont, e'_{pre\uparrow})$. This has been done since the values of $Cont\,(e'_{pre\uparrow})$ and $Cont\,(e_{pre\uparrow})$ are not equivalent for equivalent arguments $e'_{pre\uparrow}$ and $e_{pre\uparrow}$.

In $\mathcal{R}_2$ the function $Cont$ is used only in the term t

$$t = Cont\,(Current\_expression\,(\mathsf{Current\_state}))$$

in the rule STATEMENT_EXECUTION (see Section 6.3). It is easy to notice, that

- in $\mathcal{A}$ the value of $Current\_expression$ (Current_state) is $e'_{pre\uparrow}$, and the value of t is undef;
- in $\mathcal{B}$ the value of $Current\_expression$ (Current_state) is $e_{pre\uparrow}$, and the value of t is $e_{\uparrow}$.

So, the sets of applicable rules in $\mathcal{A}$ and $\mathcal{B}$ are not equal. Let S and S' be the values of Current_state in $\mathcal{A}$ and $\mathcal{B}$, respectively. Of course, in $\mathcal{A}$ in S the method N, and in $\mathcal{B}$ in S' the modified method N (see Section 7.4.3.4) are computed. Notice, that S and S' are equivalent.

Since $Cont\,(e'_{pre\uparrow}) =$ undef, the element $e'_{pre\uparrow}$ represents the last expression of N evaluated in the state S. Therefore, in the next three transitions of ASM the computation in S finishes and the value of Current_state changes into $Previous\_state\,(S)$.

Now, consider the transition of $\mathcal{B}$. During the transition from $\mathcal{R}_2(\mathcal{B})$ to $\mathcal{R}_2^{(2)}(\mathcal{B})$, the statement *'↑self'*, which is the last expression of the modified N, is evaluated. Therefore, in the next three transitions of ASM the computation in S' finishes and the value of Current_state changes into *Previous_state* (S'). Notice, that the value of *Previous_state* (S) in $\mathcal{R}_2^{(3)}(\mathcal{A})$ and the value of *Previous_state* (S') in $\mathcal{R}_2^{(5)}(\mathcal{B})$ are equivalent, so $\mathcal{R}_2^{(3)}(\mathcal{A})$ and $\mathcal{R}_2^{(5)}(\mathcal{B})$ are also equivalent. We state it formally in the following lemma.

Lemma 7.2 *Let $\tilde{C}_0^2$ and $\tilde{C}_0$ be the algebras defined above for the program P. Let T be the set defined by (7.5). Let $\tilde{C}_i^2$ and $\tilde{C}_j$ be the algebras in the computations $\mathcal{R}_2^*(\tilde{C}_0^2)$ and $\mathcal{R}_2^*(\tilde{C}_0)$, respectively, such that for some k, $k \geq 0$*

$$\tilde{C}_i^2 \equiv_{\mathcal{R}_2^{(k)}(T)} \tilde{C}_j$$

and

$$\mathcal{R}_2(\tilde{C}_i^2) \not\equiv_{\mathcal{R}_2^{(k)}(T)} \mathcal{R}_2(\tilde{C}_j).$$

Then

$$\mathcal{R}_2^{(3)}(\tilde{C}_i^2) \equiv_{\mathcal{R}_2^{(k)}(T)} \mathcal{R}_2^{(5)}(\tilde{C}_j).$$

The proof of the lemma consists of a simple analysis similar to the arguments given in Section 6.5. We leave this lemma without any proof.

As a corollary to Lemma 7.2 to we have the following theorem.

Theorem 7.5 *The computation $\mathcal{R}_2^*(\tilde{C}_0^2)$ is finite if and only if the computation $\mathcal{R}_2^*(\tilde{C}_0)$ is finite. Moreover, if both computations are finite, then the last algebras of $\mathcal{R}_2^*(\tilde{C}_0^2)$ and $\mathcal{R}_2^*(\tilde{C}_0)$ are equivalent.*

7.5.2 Equivalence between $\tilde{C}_0$ and $\mathcal{D}_0$

In this section we are going to give a proof of equivalence between two algebras $\tilde{C}_0$ and $\mathcal{D}_0$ defined for a program P. In order to give a readable proof, which skips many technical details, we are going to use slightly different notions of computation and equivalence than used in Section 5.8 and Section 6.5.

In all semantics given so far, given a program P, the notion of computation $\mathcal{R}^*(\mathcal{A})$ defined in Section 5.8 describes computation of some Abstract State Machine defined for P by a pair $\langle \mathcal{R}, \mathcal{A} \rangle$. In this section we shall use different and more general description of a computation.

By $\mathcal{R}^\infty(\mathcal{A})$ we denote an algebra, which is the fixed point of $\mathcal{R}$, if it exists. In $\mathcal{R}^\infty(\mathcal{A})$ there are two sets: EXPRESSIONS and STATES. During the computation $\mathcal{R}^*(\mathcal{A})$ the set EXPRESSIONS does not change, but STATES increases. The interpretation of STATES in $\mathcal{R}^\infty(\mathcal{A})$ will be denoted by STATES$^\infty$.

We shall consider pairs $(e,S) \in$ EXPRESSIONS $\times$ STATES$^\infty$ such that in some algebra of the computation $\mathcal{R}^*(\mathcal{A})$ the value of the term *Expression_value* (e,S) is defined. By Lemma 5.6 for each $S \in$ STATES and each $e \in$ EXPRESSIONS, e is evaluated at S at most once. Moreover, in each step of $\mathcal{R}^*(\mathcal{A})$ the term *Expression_value* (e,S) is evaluated for at most one pair (e,S). If it happens we say that the pair (e,S) is evaluated. The order in which pairs (e,S) are evaluated, is uniquely determined by $\mathcal{R}^*(\mathcal{A})$. So, we can define the following relation:

Definition 7.2 *Given a computation $\mathcal{R}^*(\mathcal{A})$, a relation $\rightarrow$ is a binary relation defined on the set* $(\text{EXPRESSIONS} \times \text{STATES}^\infty)$ *such that*

$$(e_1, S_1) \rightarrow (e_2, S_2)$$

holds if and only if there exist nonnegative k and l, $k < l$, such that

$$\begin{aligned}
&\mathcal{R}^{(k)}(\mathcal{A}) \models \\
&\quad \mathsf{Current_state} = S_1 \ \wedge \\
&\quad \mathit{Current_expression}\,(\mathsf{Current_state}) = e_1 \ \wedge \\
&\quad \mathit{Expression_value}\,(\mathit{Current_expression}\,(\mathsf{Current_state}), \mathsf{Current_state}) \neq \mathsf{undef} \ \wedge \\
&\quad \mathit{Expression_value}\,(e_2, S_2) = \mathsf{undef}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{R}^{(l)}(\mathcal{A}) \models \\
&\quad \mathsf{Current_state} = S_2 \ \wedge \\
&\quad \mathit{Current_expression}\,(\mathsf{Current_state}) = e_2 \ \wedge \\
&\quad \mathit{Expression_value}\,(\mathit{Current_expression}\,(\mathsf{Current_state}), \mathsf{Current_state}) \neq \mathsf{undef} \ \wedge \\
&\quad \mathit{Expression_value}\,(e_1, S_1) \neq \mathsf{undef}
\end{aligned}$$

and there is no pair (e_3, S_3) different from (e_1, S_1) and (e_2, S_2) such that

$$\mathcal{R}^{(k)}(\mathcal{A}) \models \mathit{Expression\_value}\,(e_3, S_3) = \mathsf{undef}$$

$$\mathcal{R}^{(l)}(\mathcal{A}) \models \mathit{Expression\_value}\,(e_3, S_3) \neq \mathsf{undef}$$

This definition states, that if

$$(e_1, S_1) \rightarrow (e_2, S_2)$$

and e_1 is evaluated at S_1, then next e_2 is evaluated at S_2. Moreover, there is no e_3 such that e_3 is evaluated at some S_3 after (e_1, S_1), but before (e_2, S_2).

We shall use the function

$$\|.\| : \text{EXPRESSIONS} \times \text{STATES}^\infty \rightarrow \text{OBJECTS}$$

defined as follows

$$\|(e, S)\| = \begin{cases} o & \exists k : \mathcal{R}_2^{(k)}(\mathcal{D}_0) \models \mathit{Expression\_value}\,(e, S) = o \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Any computation $\mathcal{R}^*(\mathcal{A})$ determines a sequence

$$\vec{\mathcal{R}}^*(\mathcal{A}) = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots \rightarrow \pi_k \rightarrow \ldots$$

where each π belongs to EXPRESSIONS $\times$ STATES$^\infty$, $e(\pi)$ and $S(\pi)$ denote the first and the second component of π, respectively, and for each $k \leq 0$

$$(e(\pi_k), S(\pi_k)) \to (e(\pi_{k+1}), S(\pi_{k+1})).$$

We write $\|\pi\|$ instead of $\|(e(\pi), S(\pi))\|$.

As in the previous theorems, we need definitions of equivalence of elements and of algebras. Let $U(\mathcal{A})$ denote the set of elements of algebra $\mathcal{A}$. By the definition of $\mathcal{D}_0$ in Section 7.4.3, $U(\mathcal{D}_0)$, besides elements which correspond to elements of $U(\tilde{\mathcal{C}}_0)$, contains some new elements.

Definition 7.3 *Let a be an element of* $U(\tilde{\mathcal{C}}_0)$*, let b be an element of* $U(\mathcal{D}_0)$*. a and b are equivalent and write*

$$a \equiv b$$

if and only if at least one of the conditions holds

(1) a and b are defined either for the same name, the same name list of names, the same name object, the same name variable, the same name method, the same name expression or the same name state in P.

(2) $a \in$ EXPRESSIONS*,* $b \in$ EXPRESSIONS*, and a represents an expression* exp *of the method N, and b represents the copy of* exp*, which is a part of the copy of N inlined into other method. Notice, that b belongs to* $\mathrm{Copy}_{\mathcal{N}}$*.*

The first condition is exactly the same as Definition 6.1. The second one defines, how elements of $Copy_{\mathcal{N}}$ relate to elements of $\tilde{\mathcal{C}}_0$. Notice, that if $a \in \tilde{\mathcal{C}}_0$ represents an expression *exp* of the method N, then by the first condition in Definition 7.3, a is equivalent to an element b of $\mathcal{D}_0$, which represents *exp*. But *exp* is a part of N, and the copy of N is inlined into another method. So, there exists the copy of *exp*, which is represented by some $c \in \mathcal{D}_0$. Hence, by the second condition, a is also equivalent to c.

We are going to consider the equivalence of $\tilde{\mathcal{C}}_0$ and $\mathcal{D}_0$ (denoted by $\tilde{\mathcal{C}}_0 \equiv_T \mathcal{D}_0$) defined by Definition 7.1 with respect to the set T defined below

$$\begin{aligned} T = S(\tilde{\mathcal{C}}_0) \setminus \{ & \\ & Temporary\_names\,(\mathcal{M}), Cont\,(e'), Name\,(e_{\mathcal{N}}), \\ & Name\,(e_{\mathcal{N}}), Receiver\,(e_{\mathcal{N}}), First\_argument\,(e_{\mathcal{N}}) \\ \}. & \end{aligned}$$

By Lemma 7.1 if a set R of update rules is applied to $\tilde{\mathcal{C}}_0$ and $\mathcal{D}_0$, and certain conditions hold, then the equivalence is preserved.

Now, we are going to prove that computations $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ and $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$ are equivalent.

Theorem 7.6 *Let P be a program, $\tilde{\mathcal{C}}_0$ be the algebra for P defined according to the definition of Section 7.4.3. Let $\mathcal{D}_0$ be the algebra defined for P according to the definition of Section 7.4.4. Assume that $\mathcal{R}_2^*(\tilde{\mathcal{C}}_0)$ and $\mathcal{R}_2^*(\mathcal{D}_0)$ are finite. Then for computations*

$$\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0) = \tilde{\pi}_0 \to \tilde{\pi}_1 \to \ldots \to \tilde{\pi}_k \to \ldots$$
$$\vec{\mathcal{R}_2}^*(\mathcal{D}_0) = \pi_0 \to \pi_1 \to \ldots \to \pi_k \to \ldots$$

and for each nonnegative k:

$$e(\tilde{\pi}_k) \equiv e(\pi_k) \wedge \|\tilde{\pi}_k\| \equiv \|\pi_k\|. \tag{7.6}$$

Proof
We proceed by induction.

Definition 7.3 gave the notion of equivalence of elements of $\tilde{\mathcal{C}}_0$ and $\mathcal{D}_0$, except the elements of STATES and BLOCKS. During the proof, we shall define inductively an equivalence relation between the elements of the sets BLOCKS and STATES in both computations.

Notice, that interpretations of BLOCKS and STATES increase during computation. Moreover, interpretation of STATES increases in different ways in the computations $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ and $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$. Suppose that the computation of the program P requires computing of M, and during computation of M some expression calls the method N. In all semantics' defined so far in order to compute method to be called, a new element of STATES is created. So, in $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ at some point of computation a state $\tilde{S}$ is created, in which M is computed. Later, another state $\tilde{S}'$ is created, in which N is being computed. In $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$ the situation is slightly different: instead of M its modification, denoted by $\tilde{M}$, is computed. In order to compute $\tilde{M}$ a new element S of STATES is created. Since all expressions of the body of N are now expressions of M, these expressions are also evaluated in S. So, evaluation of $\tilde{M}$ needs one state, while equivalent evaluation of M and N needs two states. Intentionally, computation in the states $\tilde{S}$ and $\tilde{S}'$ should be equivalent to the computation in S. Therefore, we shall define a relation of equivalence between elements of STATES in $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ and $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$ in such a way, that $\tilde{S}$ and $\tilde{S}'$ will be equivalent to S.

We give here only sketch of the proof, because its full version is long and consists of many technical details. Equivalence between some elemens will be introduced formally, and equivalence between other elements will be considered only informally.

Assume as the induction hypothesis that for each $i \leq k$

$$e(\tilde{\pi}_i) \equiv e(\pi_i) \wedge \|\tilde{\pi}_i\| \equiv \|\pi_i\|. \tag{7.7}$$

We consider several cases, depending on relationship between $S(\tilde{\pi}_k)$, $S(\pi_k)$, $S(\tilde{\pi}_{k+1})$, and $S(\pi_{k+1})$, and on types of expressions represented by $e(\pi_k)$, $e(\tilde{\pi}_k)$, $e(\pi_{k+1})$, and $e(\tilde{\pi}_{k+1})$. The proof is divided into three cases depending on whether in $S(\pi_{k+1})$ the method $\tilde{M}$ (i.e. with inlined N) is computed, or in $S(\pi_{k+1})$ a block or a method different from $\tilde{M}$ is computed.

(1) a method different form $\tilde{M}$ or a block is computed in $S(\pi_{k+1})$

(1.a) $\underline{\textit{Previous\_state}\,(S(\tilde{\pi}_{k+1})) = S(\tilde{\pi}_k) \text{ and } \textit{Previous\_state}\,(S(\pi_{k+1})) = S(\pi_k)}$

By the condition, $e(\tilde{\pi}_k)$ and $e(\pi_k)$ represent a message expression, so $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ are newly created states. The induction hypothesis implies, that the receiver and arguments of $e(\tilde{\pi}_k)$ and $e(\pi_k)$ have equivalent values in both computations. Hence, in $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ the same method P is evaluated. Since $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent the first expression of P, they are equivalent. This expression can be either a constant, a variable, or a block. By the induction hypothesis $S(\tilde{\pi}_k)$ is equivalent to $S(\pi_k)$, so we define that $S(\tilde{\pi}_{k+1})$ is equivalent to $S(\pi_{k+1})$.

(1.a.i) $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a constant: each constant expression is always evaluated to the same expression, so (7.6) holds.

(1.a.ii) $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a variable var. Then either some values have been assigned to var in both computations, and the values of var are equivalent, or no value has been assigned to var, so the value of var is nil in both computations. If $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ represent method states, it is obvious, that $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ refer to the same variable. If $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ represent block states, then by the induction hypothesis values of $Home\_state\,(S(\tilde{\pi}_{k+1}))$ and $Home\_state\,(S(\pi_{k+1}))$ are equivalent and all references to variables are equivalent, see Case (1.a.iii).

(1.a.iii) $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a block expression. In this case the sets BLOCKS in computations $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ and $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$ are extended by elements $\tilde{b}$ and b, respectively. We define that $\tilde{b}$ and b are equivalent. Since we have declared that $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ are equivalent, the values of the term $Home\_state\,(\tilde{b})$ and the term $Home\_state\,(b)$ are equivalent. Hence, if the block consists of references to variables, then in both computations any dereference of variables during computation of the block is done in the equivalent way. Moreover, the values of the terms $Previous\_state\,(Home\_state\,(\tilde{b}))$ and $Previous\_state\,(Home\_state\,(b))$ are also equivalent.

(1.b) $\underline{S(\tilde{\pi}_{k+1}) = Previous\_state\,(S(\tilde{\pi}_k)) \text{ and } S(\pi_{k+1}) = Previous\_state\,(S(\pi_k))}$

This condition implies, that $e(\tilde{\pi}_k)$ and $e(\pi_k)$ represent the last expression of a method P or a block B. By applying the same arguments as in Case (1.a), we can check that $e(\pi_{k+1})$ and $e(\tilde{\pi}_{k+1})$ represent the same expression, which can be only a message expression. By the induction hypothesis all arguments of this message expression have been evaluated to the equivalent values, and all expressions of P (or B) have been evaluated to the equivalent values. So, $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ are evaluated to the equivalent values. Hence (7.6) holds.

Since both computations continue in the previously created states $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$, by the induction hypothesis $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ are equivalent.

(1.c) $\underline{S(\tilde{\pi}_k) = S(\tilde{\pi}_{k+1}) \text{ and } S(\pi_k) = S(\pi_{k+1}) \text{ as well}}$

In this case $e(\tilde{\pi}_k)$, $e(\tilde{\pi}_{k+1})$, $e(\pi_k)$, and $e(\pi_{k+1})$ represent expressions in the same method. $e(\tilde{\pi}_{k+1})$ and $e(\pi_k)$ can be only either a constant, a variable, or a block expression. The arguments of Case (1.a) apply here. By the induction hypothesis $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ are equivalent.

(1.d) $\underline{S(\tilde{\pi}_{k+1}) \neq S(\tilde{\pi}_k),\ S(\tilde{\pi}_{k+1}) \neq Previous\_state\,(S(\tilde{\pi}_k)),}$
$\underline{S(\pi_{k+1}) \neq S(\pi_k), \text{ and } S(\pi_{k+1}) \neq Previous\_state\,(S(\pi_k))}$

This happens when $S(\tilde{\pi}_k)$ and $S(\pi_k)$ are block states, and computations in these states have been finished by a return statement. By the induction hypothesis, in both computations, blocks $\tilde{b}$ and b that have been evaluated in $S(\tilde{\pi}_k)$ and $S(\pi_k)$, respectively, are equivalent. By Case (1.a.iii) the values of the terms $\mathit{Previous\_state}\,(\mathit{Home\_state}\,(\tilde{b}))$ and $\mathit{Previous\_state}\,(\mathit{Home\_state}\,(b))$ are also equivalent. Therefore $S(\tilde{\pi}_{k+1})$ and $S(\pi_{k+1})$ are equivalent.

(2) In $S(\pi_{k+1})$ the method $\tilde{M}$ is computed

(2.a) $\underline{\mathit{Previous\_state}\,(S(\tilde{\pi}_{k+1})) = S(\tilde{\pi}_k) \text{ and } S(\pi_{k+1}) = S(\pi_k)}$

This condition holds, if in the state $S(\pi_{k+1})$ the method $\tilde{M}$ is computed. In the computation $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ in the state $S(\tilde{\pi}_k)$ the method M is computed, and $S(\tilde{\pi}_{k+1})$ is a newly created element of STATES, where evaluation of N has begun. By the induction hypothesis $S(\tilde{\pi}_k) \equiv S(\pi_k)$. We put $S(\tilde{\pi}_{k+1}) \equiv S(\pi_{k+1})$. We consider different cases depending on the type of expression represented by $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$. The element $e(\tilde{\pi}_{k+1})$ represents the first expression of "old" version of the method N, so $e(\tilde{\pi}_{k+1})$ (and $e(\pi_{k+1})$) can only represent a constant, a variable, or a block expression.

(2.a.i) $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a constant or a variable. Arguments similar to Case (1.a) work.

(2.a.ii) $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a block expression. In both computations some elements, say $\tilde{b}$ and b, are added to BLOCKS of $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ and $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$, respectively. We define that $\tilde{b}$ and b are equivalent.

(2.b) $\underline{S(\tilde{\pi}_k) = S(\tilde{\pi}_{k+1}) \text{ and } S(\pi_k) = S(\pi_{k+1})}$

(2.b.i) If $e(\pi_{k+1})$ is a constant, then (7.6) holds by the same arguments as in Case (1.a).

(2.b.ii) If $e(\pi_{k+1})$ represents a temporary variable, then since temporary variables of M are distinct from temporary variables of N, both $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ evaluate to equivalent values. If $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a global variable, then by the induction hypothesis, equivalent values have been assigned to variables named $\mathit{Name}\,(e(\tilde{\pi}_{k+1}))$ and $\mathit{Name}\,(e(\pi_{k+1}))$.

(2.b.iii) If $e(\tilde{\pi}_{k+1})$ and $e(\pi_{k+1})$ represent a block expression, then all arguments of Case (1.a) apply.

(2.c) $\underline{S(\tilde{\pi}_{k+1}) = \mathit{Previous\_state}\,(S(\tilde{\pi}_k)) \text{ and } S(\pi_k) = S(\pi_{k+1})}$

In this case in $\vec{\mathcal{R}_2}^*(\tilde{\mathcal{C}}_0)$ computation of the method N computed in the state $S(\tilde{\pi}_k)$ is completed and continued in the method M in the state $S(\tilde{\pi}_{k+1})$. In $\vec{\mathcal{R}_2}^*(\mathcal{D}_0)$ computation of the body of N inlined into M is completed. Therefore, by the construction given in Section 7.4.4, $e(\tilde{\pi}_{k+1})$ represents a message expression, and $e(\pi_{k+1})$ represents the variable var. Since N terminates by return statement, the value of its last expression $e(\tilde{\pi}_k)$ has been assigned to $e(\tilde{\pi}_{k+1})$. By construction given in Section 7.4.4, the value of $\|\pi_k\|$ is assigned to var, and e_{k+1} represents the reference to var. Since

$$\|\tilde{\pi}_k\| \equiv \|\pi_k\|$$

so we have

$$\|\tilde{\pi}_{k+1}\| \equiv \|\pi_{k+1}\|.$$

■

Comment on Case (2.a.ii):
Notice, that the values of the terms $\mathit{Previous\_state}\,(\mathit{Home\_state}\,(\tilde{b}))$ and $\mathit{Previous\_state}\,(\mathit{Home\_state}\,(b))$ are not equivalent, since

$$\mathit{Previous\_state}\,(\mathit{Home\_state}\,(\tilde{b})) = \mathit{Previous\_state}\,(S(\tilde{\pi}_k))$$

but

$$\mathit{Previous\_state}\,(\mathit{Home\_state}\,(b)) = S(\pi_k).$$

However, the value of the term $\mathit{Previous\_state}\,(\mathit{Home\_state}\,(.))$ is used only if the computation of a block in the current state ends by return statement. In Section 7.4.2 all method including blocks with return statement have been excluded from inlining.

7.6 Comments and conclusions

The methods of Smalltalk programs improvement presented in this chapter have one important limitation: by the polymorphic nature of object–oriented languages, it is uncommon that only one class of message receiver is determined. It limits both optimization techniques presented here.

However, Dean, Groove and Chambers in [DGC94] suggest a solution to this problem. It is based on a simple program transformation, which splits one polymorphic message expression into several monomorphic expressions. We show an example of this technique. Suppose that for some message expression

receiver message

we know, that its class annotation consists of two classes: Class_1 and Class_2. Then we can converse this expression into the form:

```
| var |
var ← receiver.
(var class == Class1)
    ifTrue:  [call method message from the class Class1]
    ifFalse:  [call method message from the class Class2] .
```

Of course, each such transformation can introduce an additional cost, which in the worst case makes the program run slower, or makes the program bigger. The choice of the method of optimization depends on the details of compilation into target code. The analysis of this problem is out of the scope of the thesis.

Chapter 8

Conclusions

In this thesis we have presented a definition of an operational semantics of an object–oriented programming language. Then we have given a slight simpification of this semantics. Finally we have given an application of operational semantics to a program optimisation.

8.1 Comparison with related works

Various methods of defining semantics have already been applied to object-oriented languages, including Smalltalk, and describing different aspect of such languages. The approach using lambda calculus with types allows us to study different kinds of type systems in object–oriented languages, and to investigate advantages and disadvantages. The review of such systems is presented in [AC96].

Kamin and Reddy in [KR94] present two kinds of denotational semanics of Smalltalk. The first one, called *closure model*, uses closures to encapsulate side effects on objects. The second, called *data structure model*, treats objects as records while methods are kept separated from objects.

The notion of an Abstract State Machines is appropriate to describe operational features of languages or other systems. It was used to define operational semantics of many widely used languages. Among these languages are C (see [GH93]), Prolog (see [BR94]), and Occam (see [BDR94] or [GM90]). Object-oriented languages like C++ (see [Wal95]), Prolog with objects (see [Mül94]) and Oberon (see [KP97]) have also been specified in this way. There is also a series of papers devoted to formalization of Java, and to proof of correctness of compilation and virtual machine ([BS98c, BS98a, BS98b]). In [Bör95] the reader can found a list of ASM applications in different areas of computer science.

Continuation–passing style of defining programs is applied by Appel to language ML in [App92]. There he presents several compiling's phases: creation of continuation form of a program from its abstract form, optimisation, closure conversion, and others transformations. Moreover, code generation is described.

An operational semantics of Smalltalk was given by Blakley in his doctoral thesis ([Bla92]). However, his semantics differs from our approach. First, Blakley concentrates on stack–driven model, i.e. he defines an abstract interpreter which uses a stack. It suggests some implementation details. Our approach is more abstract in the sense that we give more general description using only abstract universe STATES.

The other difference is that in order to reduce his model he considers a smaller subset of Smalltalk without class hierarchy what leads to lack of inheritance, but it allows him to develop a Hoare–style proof system for his algebras.

Appendix A

Syntax of Smalltalk

⟨Method⟩	::=	⟨Message_pattern⟩ [[⟨Temporaries⟩] ⟨Statements⟩]
⟨Message_pattern⟩	::=	⟨Unary_selector⟩
	|	⟨Binary_selector⟩ ⟨variable_name⟩
	|	[⟨Keyword⟩ ⟨variable_name⟩]$^{+}$
⟨Temporaries⟩	::=	'|' [⟨variable_name⟩]* '|'
⟨Statements⟩	::=	[⟨Statement⟩'.']* ⟨Last_statement⟩
⟨Last_statement⟩	::=	⟨Statement⟩
	|	'↑' ⟨Statement⟩
⟨Statement⟩	::=	⟨Assignments⟩ ⟨Expression⟩
⟨Assignments⟩	::=	[⟨variable_name⟩ '←']*
⟨Expression⟩	::=	⟨Primary⟩
	|	⟨Message_expression⟩
	|	⟨Cascade_message_expression⟩
⟨Primary⟩	::=	⟨Variable_name⟩
	|	⟨Literal⟩
	|	⟨Block⟩
	|	'(' ⟨Statement⟩ ')'
⟨Unary_object_description⟩	::=	⟨Primary⟩
	|	⟨Unary_expression⟩
⟨Unary_expression⟩	::=	⟨Unary_object_description⟩ ⟨Unary_message⟩
⟨Unary_message⟩	::=	⟨Unary_selector⟩

⟨*Binary_object_description*⟩ ::= ⟨*Unary_object_description*⟩
| ⟨*Binary_expression*⟩

⟨*Binary_expression*⟩ ::= ⟨*Binary_object_description*⟩ ⟨*Binary_message*⟩

⟨*Binary_message*⟩ ::= ⟨*Binary_selector*⟩ ⟨*Unary_object_description*⟩

⟨*Keyword_expression*⟩ ::= ⟨*Binary_object_description*⟩ ⟨*Keyword_message*⟩

⟨*Keyword_message*⟩ ::= **[** ⟨*Keyword*⟩ ⟨*Binary_object_description*⟩**]+**

⟨*Message_expression*⟩ ::= ⟨*Unary_expression*⟩
| ⟨*Binary_expression*⟩
| ⟨*Keyword_expression*⟩

⟨*Cascade_message_expression*⟩ ::= ⟨*Message_expression*⟩ ';' ⟨*Message*⟩
| ⟨*Cascade_message_expression*⟩ ';' ⟨*Message*⟩

⟨*Message*⟩ ::= ⟨*Unary_message*⟩
| ⟨*Binary_message*⟩
| ⟨*Keyword_message*⟩

⟨*Block*⟩ ::= '[' **[[[** ':' ⟨*Variable_name*⟩ **]***'|' **]** ⟨*Statements*⟩ **]** ']'

Terminal symbols: '|' '.' '↑' '←' '(' ')' '[' ']' ':' ';'

Nonterminal symbols: words between angle brackets, for instance ⟨*Method*⟩

Nonterminal symbols which are not defined:

⟨*Binary_selector*⟩ ⟨*Keyword*⟩ ⟨*Literal*⟩ ⟨*Variable_name*⟩ ⟨*variable_name*⟩ ⟨*Unary_selector*⟩

Auxiliary symbols used in the definition of the grammar:

::= **|** **+** ***** **[** **]**

Appendix B

Signatures of algebras

In this Appendix for each algebra introduced in this paper we give the lists of functions and rules together with the pages where these functions and rules are defined.

B.1 The algebra $\mathcal{A}_0$

LIST OF FUNCTIONS

Name of function	Domain and codomain	Page
First	LISTS_OF_NAMES → NAMES	25
But_first	LISTS_OF_NAMES → LISTS_OF_NAMES	25
empty	LISTS_OF_NAMES	25
Class	OBJECTS → CLASSES	26
nil	OBJECTS	27
Variable_value	VARIABLES → OBJECTS	28
Global_variable	SHARED_VARIABLE_NAMES → VARIABLES	28
Class_variable	SHARED_VARIABLE_NAMES × CLASSES → VARIABLES	28
Pool_variable	SHARED_VARIABLE_NAMES × NAMES → VARIABLES	28
Object_variable	NAMES × OBJECTS → VARIABLES	28
Pseudovariable	NAMES → VARIABLES	28
Shared_pools	CLASSES → LISTS_OF_NAMES	29
self	NAMES	30
super	NAMES	30
Method	NAMES × CLASSES → METHODS	31
Method_class	METHODS → CLASSES	31
Argument_names	METHODS → LISTS_OF_NAMES	32
Temporary_names	METHODS → LISTS_OF_NAMES	32
First_statement	METHODS → EXPRESSIONS	32
Superclass	CLASSES → CLASSES	33
Name	EXPRESSIONS → NAMES	35

Argument_names	Expressions → Lists_of_names	35
Constant_value	Expressions → Objects	35
First_statement	Expressions → Expressions	35
Receiver	Expressions → Expressions	36
First_argument	Expressions → Expressions	36
Next_argument	Expressions → Expressions	36
Superexpression	Expressions → Expressions	36
Assignments	Expressions → Lists_of_names	36
Next_statement	Expressions → Expressions	36
Return_prefix	Expressions → Names	36
Current_expression	States → Expressions	49
Method_class	States → Classes	49
Receiver	States → Objects	49
Temporary_variable	Names × States → Variables	50
Expression_value	Expressions × States → Objects	50
Current_state	States	50
Previous_state	States → States	50
Home_state	States → States	50
block	Classes	51
First_statement	Blocks → Expressions	51
Argument_names	Blocks → Lists_of_names	51
Home_state	Blocks → States	51
Superfluous_state	States	52
Assignments	Lists_of_names	52
Class	Classes	52
Return_prefix	Names	52
Last_value	Objects	52
Argument	Expressions	52
Argument_names	Lists_of_names	52
Temporary_names	Lists_of_names	52
Variable	Variables	52
Variable_name	Variables	52
Pool_names	Lists_of_names	52
Method	Methods	53
Receiver	Objects	53

List of rules

Name of rule	Page
MAIN_RULE	56
STATE_CREATION	57
METHOD_STATE_CREATION	58
NEW_METHOD_STATE	59
METHOD_VARIABLE_INITIALIZATION	59
METHOD_ARGUMENT_PASSING	59

B.2 The algebra $\mathcal{B}_0$

LIST OF FUNCTIONS

LIST OF RULES

B.3 The algebras $\mathcal{C}_0$ and $\mathcal{D}_0$

LIST OF FUNCTIONS

Name of function	Domain and codomain	Page
First	CLASS_LISTS → CLASSES	103
TypesOf	EXPRESSIONS → CLASS_LISTS	103
Method	EXPRESSIONS → METHODS	103

LIST OF RULES

Name of rule	Page
MODIFIED_LOOKUP	104
EXPRESSION_EVALUATION	105

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[AH95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. Technical Report TRCS 95-04, Computer Science Department, University of California, Santa Barbara, March 1995.

[ANS97] ANSI J20 technical committee. *Draft American National Standard for Information Systems – Programming Languages – Smalltalk*, December 1997. `http://www.stic.org/STLang/F002.htm`.

[App92] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers– Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[BDR94] E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.

[Bla92] B. Blakley. *A Smalltalk Evolving Algebra and its Uses*. PhD thesis, University of Michigan, Ann Arbor, Michigan, 1992.

[Bör95] E. Börger. Annotated Bibliography on Evolving Algebras. In E. Börger, editor, *Specification and Validation Methods*, pages 37–51. Oxford University Press, 1995.

[BR94] E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.

[BS98a] E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.

[BS98b] E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In L. Brim and J. Gruska and J. Zlatuska, editor, *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic*, number 1450 in LNCS. Springer, August 1998.

[BS98c] E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[DGC94] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Technical Report TR-94-12-01, University of Washington, Department of Computer Science and Engineering, December 1994.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Fundamentals of Interactive Computer Graphics*. The Systems Programming Series. Addison-Wesley, Reading, MA, USA, second edition, 1990.

[GH93] Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In ACM, editor, *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages, January 17–19, 1990, San Francisco, CA*, pages 136–150, New York, NY, USA, 1990. ACM Press.

[GM90] Y. Gurevich and L. Moss. Algebraic Operational Semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 176–192. Springer, 1990.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1979.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80 The Language*. Addison–Wesley, Reading (MA), 1989.

[Gur91] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[Gur00] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 2000. to appear.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In P. America, editor, *Proceedings of the ECOOP '91 European Conference on Object-oriented Programming*, LNCS 512, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[HP79] Matthew Hennessy and Gordon D. Plotkin. Full Abstraction for a Simple Programming Language. In J. Bečvář, editor, 8^{th} *Symposium on Mathematical Foundations of Computer Science*, volume 74 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1979.

[KMW99] Antoni Kościelski, Marcin Młotkowski, and Jerzy Witkowski. Semantics of the Smalltalk Programming Language. Submitted to Theory and Practice of Object Systems (TAPOS), 1999.

[KP97] P. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.

[KR94] Samuel N. Kamin and Uday S. Reddy. Two semantics models of object–oriented languages. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object–Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. The MIT Press, 1994.

[Mül94] B. Müller. A Semantics for Hybrid Object–Oriented Prolog Systems. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994.

[NBB+60] Peter Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. MacCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language Algol 60. *Numerische Mathematik*, 2:106–136, 1960.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. Technical Report DAIMI PB - 345, Computer Science Department, Aarhus University, March 1991.

[Rey81] John C. Reynolds. The essence of ALGOL. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland, Amsterdam, 1981.

[Str66] C. Strachey. Towards a formal semantics. In *IFIP TC2 Working Conference on Formal Language Description Languages for Computer Programming*, pages 198–220, Amsterdam, 1966. North-Holland.

[Str91] Bjarne Stroustrup. *The C++ Programming Language: Second Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1991.

[US91] David Ungar and Randall B. Smith. SELF. The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–205, July 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

[Wal95] C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.