Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

A. V. Zamulin

A STATE-BASED SEMANTICS OF A PASCAL-LIKE LANGUAGE

Preprint
104

Novosibirsk 2003

A formal model of a Pascal-like language with storable locations is presented. A storable location can both store values and be stored as a value in another location. Such a location is part of the state of an imperative program. An access function is associated with a set of locations of the same type. An update of an access function causes the update of the content of the corresponding location. The parameter of a function or procedure declared as a reference parameter accepts only locations as arguments so that a value associated with the location can be updated by the procedure. In this way, the mechanism of call-by-reference in addition to the mechanism of call-by-value is modeled. Formal semantics of some typical statements and function/procedure definition is given in the paper.

Keywords: formal semantics, formal model, imperative language, Abstract State Machines, formal methods, storable locations.

Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

А. В. Замулин

ФОРМАЛЬНАЯ МОДЕЛЬ ПАСКАЛЕПОДОБНОЙ ПРОГРАММЫ

Препринт
104

Новосибирск 2003

Представлена формальная модель Паскалеподобного языка с хранимыми ячейками. Хранимая ячейка может как хранить значение, так и храниться сама как значение в другой ячейке. Такая ячейка является частью состояния императивной программы. С множеством однотипных ячеек связывается функция выборки. Модификация этой функции вызывает обновление состояния соответствующей ячейки. Параметр процедуры, объявленный как параметр-ссылка, замещается ячейкой при ее вызове, так что содержимое этой ячейки может быть изменено данной процедурой. Таким путем моделируется механизм подстановки ссылкой, дополняющий механизм подстановки значением. В препринте определена также формальная семантика типичных операторов и описания процедуры Паскалеподобного языка программирования.

1. INTRODUCTION

Abstract State Machines [5, 7] are now widely used for the formal definition of programming languages [6, 8, 9, 15, 16, 19]. An Abstract State Machine (ASM) is usually a state represented by an algebra and a transition rule reminding a statement of an imperative language. The algebra consists of a *superuniverse* subdivided into *universes* and a number of co-called *dynamic functions*. Transition rules are recursively built from function updates and the skip rule by a number of rule constructors. The interpretation of a transition rule generally causes an update of the current state either by modifying a dynamic function or inserting a new element into a universe.

With the use of this technique, the semantics of a programming language is usually defined by the work of an abstract interpreter whose behavior is expressed in terms of transition rules. A number of dynamic functions are declared in the interpreter, representing its state. The semantics of a program component (expression, statement, etc.) is described by a corresponding transition rule, which indicates in what way one or another dynamic function should be updated in the process of execution of the component.

In this paper, we propose another way of defining the semantics of an imperative programming language with the use of ASMs. Instead of using transition rules, we suggest to construct, for each programming language, a particular abstract model based on the ideas of ASMs. When such a model is constructed, the semantics of a program component is defined in the terms of the model.

We concentrate on modeling program components involving operations with locations. In the traditional ASMs, transformations of the state are performed by means of *update sets*. Each update in the set is a pair (loc, val), where *loc* is a location and *val* a value to be associated with it. A location is a pair $(f, \langle a_1, ..., a_n \rangle)$ representing an n-ary function f applied to a tuple of elements $\langle a_1, ..., a_n \rangle$, the value *val* is then the value to be produced by $f(a_1, ..., a_n)$ in the new state. In this approach, a location is not a value that can be used and/or produced by another function. It can only be considered as the name of a value.

At the same time, locations as values are typical of modern imperative languages. For example, a location value (l-value of C) is used in the left-hand side of the assignment statement. Pointers are another example of

[1]This research is supported in part by Russian Foundation for Basic Research under Grant 01-01-00787.

location values. Thus, if $loc(t)$ is the type of locations of type t, then program variables declared in C as $t\ x$ and $t^*\ p$ can be considered as having respective types $loc(t)$ and $loc(loc(t))$. When a location is used in the right-hand side of the assignment statement or as value parameter in the function call, it is first implicitly dereferenced (i.e., the value stored in the location is extracted); any further dereferencing is explicit. For instance, p is implicitly dereferenced to produce a value of type t^* and then explicitly dereferenced to produce a value of type t. The strict rules of implicit dereferencing permit the compiler to correctly type-check a program and generate the correct code. Locations as values also permit another kind of function parameter substitution, *call-by-reference*. In this case a location itself is passed as argument to a function, and its content can be updated by the function (although this kind of parameters is discarded in such a modern language as Java [2], it is reintroduced in C# [11]).

Thus, our purpose is to suggest a state model with *storable locations*, i.e., locations that can both store values and be stored as a value in other locations. In this way, a state model close to that of an imperative programming language can be designed. A dynamic *access function* should be associated with each location type. An update of the access function must lead to the update of the corresponding location. A function parameter declared as a reference parameter must serve as an alias of an argument location so that a value associated with the location can be updated by the function. So, formal definition of *call-by-reference* parameter substitution is another purpose of this work.

Several works have already been devoted to formal treatment of addresses, pointers and call-by-reference parameter substitution. The majority of them refer to demonstration of a definition method to a specifically constructed language. For example, a number of axioms are devised in [1] for operations that allocate and dispose memory. A simple memory model is used in the work: stack variables are not locations (and therefore cannot be passed as reference arguments to a function), heap locations contain records whose fields are not locations (and therefore a field cannot be passed to a function as a reference argument), the problems of function definitions are not considered at all. Similarly, an extension of Hoare logic that permits reasoning about programs using shared data structures is proposed in [14]. Once again, a very simple memory model capturing the low-level character of machine language (for example, one cannot have the address of an array or record and pass it as argument to a function) is proposed. An attempt

to define pointers by means of the algebra of partial maps has been undertaken in [12]. However, the problem has not been tightly coupled with the problem of parameter substitution, which is our aim.

The paper is organized as follows. A way of constructing a program signature is described in Section 2. A program algebra of a given program signature is elaborated in Section 3. A formal notion of the program is presented in Section 4. Examples of expression and statement construction and their interpretation in a program are given in Sections 5 and 6 respectively. Formal semantics of the function definition and constant definition is presented in Section 7. Some conclusions and directions of further work are outlined in Section 8.

2. THE PROGRAM SIGNATURE

2.1. The type system

The program model described in this paper is based on a simple type system with the following grammar:

$$\mathcal{T} ::= \texttt{BASE} \mid \mathit{array}(\texttt{BASE}, \texttt{BASE}) \mid \mathit{loc}(\mathcal{T}), \tag{1}$$

where `BASE` is a set of primitive types, *array* a Pascal-like array type constructor and *loc* a location (pointer) type constructor. The elements of $\mathcal{T}$ are called *types*. In the sequel, $\mathcal{T}^*$ stands for the set of all sequences with an element of a sequence being either t or $\mathit{ref}\ t$, where $t \in \mathcal{T}$, and $\mathcal{T}^v$ stands for $\mathcal{T} \cup \{\texttt{void}\}$, where `void` is a special type not belonging to $\mathcal{T}$ and possessing a single value $\perp$.

So, our type system includes the part `BASE` with the signature $\Sigma_{dat} = (S_{dat}, F_{dat})$, which defines primitive data types (sorts and operations) using the facilities of an algebraic specification language. We do not describe a mechanism for the specification of this part of the language. Any specification language whose semantics is given as a class of many-sorted algebras is suitable for this purpose.

We also do not provide array types and location types with their own operations. All needed operations will be introduced as special kinds of expressions. This is caused by the fact that the operations with pointer and arrays variables (as well as with many other structured variables) in programming languages normally produce locations, and algebraic specifications are not well suited for dealing with this kind of data. For brevity, we consider only arrays of elements of primitive types.

2.2. The program schema

The program schema defines symbols used in a program. It is formed on the base of variable, constant and function declarations. Let IDENT be an enumerable non-empty set representing *identifiers*, then the *program schema* $\mathcal{S}$ over the type system $\mathcal{T}$ consists of:

- a finite set of array types AT,
- a finite set of location types LT,
- three finite mappings var : IDENT $\longrightarrow \mathcal{T}$, $const$: IDENT $\longrightarrow$ BASE, and $func$: IDENT $\times \mathcal{T}^* \longrightarrow \mathcal{T}^v$

such that:

- if $loc(t) \in LT$, then $t \in (\texttt{BASE} \cup AT \cup LT)$,
- if $x \in$ IDENT, then either $var(x)$ or $const(x)$ may be defined and its result is a type belonging to $\texttt{BASE} \cup AT \cup LT$.

Thus, only the types defined in $\mathcal{S}$ can be used for the definition of location types and declaration of variable and constants, and there may be no variable and constant with the same identifier. Note that constants are not provided with values and functions are not provided with bodies in the program schema, they are part of the *program specification* discussed later.

In the sequel, the elements of the mappings $var(x)$ and $const(x)$ are denoted by pairs (x, t), and the elements of the mapping $func$ are denoted by triples (f, r, t), where r is a sequence of parameter types. Functions with the result type void are called *procedures*.

The program schema $\mathcal{S}$ naturally defines the signature $\Sigma = (S, F)$ extending the signature $\Sigma_{dat} = (S_{dat}, F_{dat})$ as follows:

- $S = S_{dat} \cup AT \cup LT \cup \{loc(t) \mid \forall x, t.(x, t) \in var\}$;
- $F = F_{dat} \cup var \cup const \cup func \cup F_{acc}$, where F_{acc} is a set including an *access function* symbol $\_ \uparrow_{loc(t),t}$ for each $loc(t) \in S$.

Thus, a location type is created for each variable in the schema, and an access function symbol is created for each location type.

3. PROGRAM ALGEBRA

A program algebra is a counterpart of the state of an object program. It contains basic data type operations and constant and variable locations in static (stack) or dynamic (heap) memory. Each of them has a static part (basic algebra) mainly consisting of the basic data types and a dynamic part mainly consisting of variable locations. The static and dynamic parts together are a state algebra representing a program state.

3.1. Basic algebra

A *basic algebra* $\mathtt{B}$ associates with each type t a set $\mathtt{B_t}$, its *carrier*, and with each operation of t a partial function, its *implementation*. The carriers of our types are defined as follows:

- the carrier of each primitive type t is a set $\mathtt{B_t}$. The carriers of primitive types are assumed to be pairwise disjoint;
- the carrier of each $loc(t)$ is a special set $\mathtt{B_{Locs}}$ disjoint from all basic types carriers; elements of $\mathtt{B_{Locs}}$ are *locations*.

We also assume the existence of the value `nil` that does not belong to any carrier.

Each primitive type is supposed to be endowed with some operations, and each of them is implemented as a function $\mathtt{op^B} : \mathtt{B_{t_1}} \times \cdots \times \mathtt{B_{t_n}} \longrightarrow \mathtt{B_t}$ when $n > 0$, otherwise $\mathtt{op^B} \in \mathtt{B_t}$. The only predefined operation on locations is comparison operation "=" such that $\mathtt{l} = \mathtt{l'}$ is true iff both $\mathtt{l}$ and $\mathtt{l'}$ are the same element of $\mathtt{B_{Locs}}$. As it was mentioned above, array and location types are not provided with predefined operations.

3.2. State algebra

A state algebra represents a program state. In what follows, $\mathtt{B}$ is a base algebra and $\Sigma = (S, F)$ a program signature. A *state algebra* $\mathtt{A}$ of signature $\Sigma = (S, F)$, called a Σ-state in the sequel, is created in the following way:

- $\mathtt{A_t} = \mathtt{B_t}$ for any primitive type t.
- A finite set $\mathtt{A_{loc(t)}} = \mathtt{A^\circ_{loc(t)}} \cup \{\mathtt{nil}\}$, where $\mathtt{A^\circ_{loc(t)}} \subset \mathtt{B_{Locs}}$, is associated with each type $loc(t)$ so that $\mathtt{A^\circ_{loc(t)}} \cap \mathtt{A^\circ_{loc(t')}} = \emptyset$ for two different types t and t';
- a set of finite injective mappings $\mathtt{A_{ind}} \rightarrow \mathtt{A^\circ_{loc(t)}}$ is associated with each array type $array(ind, t)$;
- an element, $\mathtt{c^A} \in \mathtt{A_t}$, is associated with each constant declaration (c, t);
- an element, $\mathtt{x^A} \in \mathtt{A^\circ_{loc(t)}}$, called *location constant*, is associated with each variable declaration (x, t);
- a partial function $\_!^{\mathtt{A}} : \mathtt{A_{loc(t)}} \rightarrow \mathtt{A_t}$ is associated with each $\_!_{loc(t),t} \in F_{acc}$ so that $\mathtt{x^A} \uparrow$ is defined for each location constant $\mathtt{x^A}$ in $\mathtt{A}$.

In this way, each location type is provided with a number of locations and a special element `nil`, each constant is provided with a value, and each variable is provided with an initialized location. Note that, according to the last item in the above definition, an array variable will be provided with a

location initialized with a corresponding finite mapping. In this way, both an array variable and an array element are provided with a location.

A state algebra is well-formed if:

- each location constant is supplied with a different location,
- the range of the finite mapping associated with an array variable does not intersect with both the range of the finite mapping associated with any other array variable and the set of locations assigned to variables.

It is clear that in a well-formed state algebra a location constant is a real counterpart of a program variable having an address different from the addresses of all the other variables, and a finite mapping associated with an array is a counterpart of a program array where elements have different addresses and there is a unique address for each index value (note that the mappings are injective).

A location `l` is said to be *allocated* in a state `A` if $\texttt{l} \in \texttt{A}_{\texttt{loc(t)}}$ for some $loc(t) \in S$. Locations associated with variable identifiers and array components are part of the static memory, other locations that may be dynamically created are part of the dynamic memory (heap). The set of all allocated locations in `A` is denoted by $\texttt{A}_{\texttt{loc}}$ in the sequel.

If `l` and `a` are elements of respective sorts $loc(t)$ and t, such that $\texttt{l}\uparrow = \texttt{a}$, then `a` is the *content* of the location `l` (`a` is *stored* in `l`). The application of the function "$\uparrow$" to its argument `l` is called the *dereferencing* of `l`.

A location sort and access function can be different in different states. This reflects the fact that locations can be dynamically created in a heap. However, different Σ-states must have the same base algebra. We denote the set of all Σ-states with the same base `B` by $\texttt{state}_{\texttt{B}}(\Sigma)$ and mean by a Σ_B-state a Σ-state with the base algebra `B`. The set of elements associated with a type in a state algebra is called a *sort* in the sequel.

3.3. State updates

One state can be transformed into another by a *state update*, which is either a *location update* or *sort update*. The first one serves for an update of a program variable and the other one serves for extending a location sort with a new location.

Definition 1. A *location update* in a Σ_B-state `A` is a triple `(loc(t), l, a)`, where `t` is a type from S different from an array type, `l` an element of sort $\texttt{A}_{\texttt{loc(t)}}$, and `a` an element of sort $\texttt{A}_{\texttt{t}}$. ■

A location update $\alpha = (\mathtt{loc(t)}, \mathtt{l}, \mathtt{a})$ serves for transformation of a Σ_B-state $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}\alpha$ in the following way:

- $\mathtt{g}^{\mathtt{A}\alpha} = \mathtt{g}^{\mathtt{A}}$ for any function symbol g in F different from $\_!_{loc(t),t}$;
- $\mathtt{x}^{\mathtt{A}\alpha} = \mathtt{x}^{\mathtt{A}}$ for any constant or variable name x in F;
- $(\mathtt{l}!)^{\mathtt{A}\alpha} = \mathtt{a}$;
- $(\mathtt{l}'!)^{\mathtt{A}\alpha} = (\mathtt{l}'!)^{\mathtt{A}}$ for any $\mathtt{l}' \in \mathtt{A}_{\mathtt{loc(t)}}$ different from $\mathtt{l}$;
- $\mathtt{A}\alpha_{\mathtt{t}} = \mathtt{A}_{\mathtt{t}}$ for any $t \in S$.

Following Gurevich [5], we say that $\mathtt{A}\alpha$ is obtained by firing the update α on $\mathtt{A}$, which changes the content of a location (and nothing else). In this way, a variable defined in the program schema may be updated. Note that an array location cannot be updated, i.e., it cannot be supplied with a different finite mapping (however, an array element location may be updated).

Definition 2. A *sort update* δ in a Σ_B-state $\mathtt{A}$ is a pair $(\mathtt{loc(t)}, \mathtt{l})$, where $\mathtt{l}$ is an element such that $\mathtt{l} \in \mathtt{B}_{\mathtt{Locs}}$ and $\mathtt{l} \notin \mathtt{A}_{\mathtt{loc}}$. ■

A sort update $\delta = (\mathtt{loc(t)}, \mathtt{l})$ transforms a Σ_B-state $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}\delta$ in the following way:

- $\mathtt{A}\delta_{\mathtt{loc(t)}} = \mathtt{A}_{\mathtt{loc(t)}} \cup \{\mathtt{l}\}$,
- $\mathtt{A}\delta_{\mathtt{loc(t')}} = \mathtt{A}_{\mathtt{loc(t')}}$ for any $t' \in S$ different from t,
- $\mathtt{f}^{\mathtt{A}\delta} = \mathtt{f}^{\mathtt{A}}$ for any f in F.

Thus, the sort update $\delta = (\mathtt{loc(t)}, \mathtt{l})$ extends the set of elements of a certain location sort by a new element different from any location existing in $\mathtt{A}$. A set of location/sort updates is called an *update set*. The update set is *inconsistent* if it contains

- two location updates $\alpha_1 = (\mathtt{loc(t)}, \mathtt{l}, \mathtt{a})$ and $\alpha_2 = (\mathtt{loc(t)}, \mathtt{l}, \mathtt{a}')$ s.t. $\mathtt{a} \neq \mathtt{a}'$ (two location updates differently defining an access function for the same location), or
- a $\delta_1 = (\mathtt{loc(t)}, \mathtt{l})$ and $\delta_2 = (\mathtt{loc(t')}, \mathtt{l})$ and $\mathtt{t} \neq \mathtt{t}'$ (there is an attempt to use one and the same location in different location sorts);

the update set is *consistent* otherwise. A consistent update set Γ applied to a Σ_B-state $\mathtt{A}$ transforms $\mathtt{A}$ into a new Σ_B-state $\mathtt{A}'$ by simultaneous firing all $\alpha \in \Gamma$ and all $\delta \in \Gamma$. If Γ is inconsistent, the new state in not defined. If Γ is empty, $\mathtt{A}'$ is the same as $\mathtt{A}$. We consider only consistent update sets in the sequel and denote the application of Γ to a state $\mathtt{A}$ by $\mathtt{A}\Gamma$.

The *sequential union* of two update-sets Γ_1 and Γ_2, denoted by $\Gamma_1;\Gamma_2$, is a consistent update set created as follows: delete from $\Gamma_1 \cup \Gamma_2$ any $\alpha_1 \in \Gamma_1$ for which there is an $\alpha_2 \in \Gamma_2$, such that $\{\alpha_1, \alpha_2\}$ is inconsistent.

4. PROGRAM

The model of a program defined in this section is a counterpart of an object program possessing a number of states and a number of functions and procedures manipulating the state. In what follows, the set of all consistent update sets in a $\Sigma_{\mathtt{B}}$-state $\mathtt{A}$ is denoted by $\Gamma\Gamma^{\mathtt{A}}(\Sigma)$. For any $\Sigma_{\mathtt{B}}$-state $\mathtt{A}$ and any sequence of types $r = t_1 \dots t_n$, we denote $\mathtt{A}_{\mathtt{t}_1} \times \cdots \times \mathtt{A}_{\mathtt{t}_n}$ by $\mathtt{A}_{\mathtt{r}}$, which is a singleton set if $n = 0$. We also introduce a notion of a pair $\langle \Gamma, \mathtt{v} \rangle$, where Γ is an update set (possibly empty) and $\mathtt{v}$ an algebra element. The set $\Gamma\Gamma^{\mathtt{A}}_{\mathtt{t}}(\Sigma)$ is the set of all pairs $\langle \Gamma, \mathtt{v} \rangle$ such that $\mathtt{v} \in \mathtt{A}_{\mathtt{t}}$ and $\Gamma \in \Gamma\Gamma^{\mathtt{A}}(\Sigma)$.

A *program* $\mathtt{P(B)}$ over a program signature Σ consists of:

1. A subset $|\mathtt{P(B)}|$ of $\mathtt{state_B}(\Sigma)$ called the *carrier* of $\mathtt{P(B)}$ (each $\mathtt{A} \in |\mathtt{P(B)}|$ is a *program state*).

2. For each $(f, r, t) \in \mathit{func}$, where $r = u_1, \cdots, u_n$ and u_i is either t_i or $\mathit{ref}\ t_i$, and each $\mathtt{A} \in |\mathtt{P(B)}|$, a partial function

$$\mathtt{f}^{\mathtt{A}}_{\mathtt{r}} : \mathtt{A}_{\mathtt{u}_1} \times \ldots \times \mathtt{A}_{\mathtt{u}_n} \to \Gamma\Gamma^{\mathtt{A}}_{\mathtt{t}}(\mathcal{S}),$$

where $\mathtt{A}_{\mathtt{u}_i}$ is $\mathtt{A}^{\circ}_{\mathtt{loc}(\mathtt{t}_i)}$ if u_i is $\mathit{ref}\, t_i$ and $\mathtt{A}_{\mathtt{t}_i}$ in the opposite case.

Thus, a program possesses a number of states, and, in each state, a function associated with the corresponding function declaration. Each function uses a location sort $\mathtt{A}^{\circ}_{\mathtt{loc}(\mathtt{t})}$ for a parameter declared as $\mathit{ref}\, t$ to provide call by reference substitution. Note that functions associated with the same function declaration may be different in different states (for instance, because they depend on the content of global variables).

The carrier $|\mathtt{P(B)}|$ must satisfy the following program invariant: any constant (including a location constant) and a mapping associated with an array variable must be the same in any $\mathtt{A} \in |\mathtt{P(B)}|$. This is a natural requirement stating that a constant may not have different values and neither a variable nor an array element may be provided with different locations in different program states. Note that the location update described in the previous section cannot violate the program invariant.

5. EXPRESSIONS

Now we introduce several rules for creating some typical expressions of a Pascal-like language. They extend the usual rules of expression (term) construction, which are part of the base of recursion.

The rules of the function call evaluation normally state that each next argument in a list of function arguments is evaluated after the previous one and the evaluation of both the argument and the function may change the state. Therefore, given a program signature Σ, a program $\mathtt{P(B)}$ over Σ and a program state $\mathtt{A} \in |\mathtt{P(B)}|$, the interpretation of an expression $f(e_1, \cdots, e_n)$ produces a pair $\langle \Gamma, \mathtt{v} \rangle \in \Gamma\Gamma^{\mathtt{A}}_{\mathtt{t}}(\Sigma)$. The first component of this pair is the empty set when neither the function nor any of the expressions $e_1, ..., e_n$ produces an update set (i.e., does not change the state). For this reason, the interpretation of any expression is defined in terms of the pairs. The interpretation of an expression e in the state $\mathtt{A}$ with the base $\mathtt{B}$ is denoted by $[\![e]\!]^{\mathtt{A(B)}}$.

1. If (x, t) is a variable declaration, then x is an expression of type $loc(t)$.
Interpretation: $[\![x]\!]^{\mathtt{A(B)}} = \langle \emptyset, \mathtt{x}^{\mathtt{A}} \rangle$. Thus, the location assigned to x is produced as the second element of the pair, and the first element is the empty update set.

2. If $(p, loc(t))$ is a location (pointer) variable declaration, then $p \uparrow$ is an expression of type $loc(t)$.
Interpretation: $[\![p \uparrow]\!]^{\mathtt{A(B)}} = \langle \emptyset, \mathtt{p}^{\mathtt{A}} \uparrow_{\mathtt{loc(loc(t)),loc(t)}} \rangle$. Thus, the location stored in p is produced as the second element of the pair, and the first element is the empty update set. The interpretation of the expression is not defined if $\mathtt{p}^{\mathtt{A}} = \mathtt{nil}$.

3. If (f, r, t) is a function declaration, where $r = u_1, ...u_n$ and u_i is either t_i or $ref\ t_i$, and

$$e_i \text{ is } \begin{cases} \text{an expression of type } loc(t_i) \text{ when } u_i \text{ is } ref\ t_i, \\ \text{an expression either of type } t_i \text{ or } loc(t_i) \text{ in the opposite case,} \end{cases}$$

then $f(e_1, ..., e_n)$ is an expression of type t called a *function call*.
Interpretation: Let $\mathtt{A_0}$ be a program state, then

$$[\![f(e_1, \ldots, e_n)]\!]^{\mathtt{A_0(B)}} = \langle \Gamma, \mathtt{v} \rangle,$$

where $\Gamma = \Gamma_1; \ldots; \Gamma_{n+1}$, $\mathtt{v} = \mathtt{snd}(\mathtt{f}^{\mathtt{A_n}}_{\mathtt{r}}(\mathtt{v_1}, \ldots, \mathtt{v_n}))$, $\Gamma_1 = \mathtt{fst}([\![e_1]\!]^{\mathtt{A_0(B)}})$, $\mathtt{A_1} = \mathtt{A}\Gamma_1$, $\ldots$, $\Gamma_n = \mathtt{fst}([\![e_n]\!]^{\mathtt{A_{n-1}(B)}})$, $\mathtt{A_n} = \mathtt{A_{n-1}}\Gamma_n$,

$$\mathtt{v_i} = \begin{cases} \mathtt{snd}([\![e_i]\!]^{\mathtt{A_{i-1}(B)}}) & \text{if } u_i \text{ is } ref\ t_i \\ & \text{and } e_i \text{ is an expression of type } loc(t_i) \\ \mathtt{snd}([\![e_i]\!]^{\mathtt{A_{i-1}(B)}}) & \text{if } u_i \text{ is } t_i \\ & \text{and } e_i \text{ is an expression of type } t_i, \\ \mathtt{snd}([\![t_i \uparrow]\!]^{\mathtt{A_{i-1}(B)}}) & \text{if } u_i \text{ is } t_i \\ & \text{and } e_i \text{ is an expression of type } loc(t_i), \end{cases}$$

and $\Gamma_{n+1} = \mathtt{fst}(\mathtt{f}_r^{\mathtt{A}_n}(\mathtt{v}_1, \ldots, \mathtt{v}_n))$, provided each e_i, $i = 1, \ldots, n$, is defined in $\mathtt{A}_i$, and $\mathtt{f}_r^{\mathtt{A}_n}$ is defined for the tuple $(\mathtt{v}_1, \ldots, \mathtt{v}_n)$; $[\![f(e_1, \ldots, e_n)]\!]^{\mathtt{A}_0(\mathtt{B})}$ is undefined otherwise.

Thus, the interpretation of a function call in $\mathtt{A}$ causes the invocation of the function associated with f in $\mathtt{A}_n$. An argument is dereferenced when a location is substituted where its content is needed, and it is directly substituted in all other cases.

4. If ar is an expression of type $loc(array(ind, t))$ and i an expression either of type ind or $loc(ind)$, then $ar[i]$ is an expression of type $loc(t)$.
Interpretation: $[\![ar[i]]\!]^{\mathtt{A}(\mathtt{B})} = [\![ar \uparrow (i \uparrow)]\!]^{\mathtt{A}(\mathtt{B})}$ if i has type $loc(ind)$ and $[\![ar[i]]\!]^{\mathtt{A}(\mathtt{B})} = [\![ar \uparrow (i)]\!]^{\mathtt{A}(\mathtt{B})}$ if i has type ind.

6. STATEMENTS

Each statement is described in two parts: definition and semantics. The *definition* generally states what a well-formed statement is. The *semantics* indicates the kind of the statement, the type of the result that can be produced, the update set, and value that can be produced by the execution of the statement.

A statement can be provided with a label used in a goto statement for control transfer. A label is visible only in the block where it is declared, i.e., one cannot transfer control into a block or a function body, one cannot also leave the function body by the goto statement or jump from one branch of the conditional statement to another branch. However, one can transfer control from an inner block into an outer block.

We assume in this section that Σ_0 is a program signature and any other signature is an extension of Σ_0. The semantics of a statement St is denoted in the same way as the semantics of an expression, i.e. $[\![St]\!]^{\mathtt{A}(\mathtt{B})}$ means execution of St in the state $\mathtt{A}$ with the base $\mathtt{B}$.

6.1. Results of statement inspection and execution

Taking into account the fact that some statements do not transfer control and some others do, we distinguish three kinds of statement: ordinary, break and return statements. For this reason, the semantics of a statement produces a quadruple $\langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$, where $\mathtt{t}$ is a type, $\mathtt{k}$ is one of the symbols **O**, **B**, or **R** denoting, respectively, an *ordinary*, *goto* or *return* statement, Γ is an update set and $\mathtt{v}$ is a value of the type $\mathtt{t}$. The value $\mathtt{v}$ may be either the label of the statement indicated in the goto statement or the value produced

by the return statement (the value $\perp$ if no value is produced).

The sequential union operation over update sets is extended to the corresponding operation on the quadruples $\langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$ in the following way:

$$\Gamma;\langle \mathtt{t}, \mathtt{k}, \Gamma', \mathtt{v} \rangle = \langle \mathtt{t}, \mathtt{k}, (\Gamma;\Gamma'), \mathtt{v} \rangle.$$

It is not difficult to prove that the operation is both left-associative and right-associative, i.e.,

$$(\Gamma_1;\Gamma_2);\langle \mathtt{t}, \mathtt{k}, \Gamma_3, \mathtt{v}_3 \rangle = \Gamma_1;(\Gamma_2;\langle \mathtt{t}, \mathtt{k}, \Gamma_3, \mathtt{v}_3 \rangle).$$

For this reason, we will omit parentheses when several tuples are sequentially united. The result of the infinite sequence of the sequential union operations

$$\Gamma_1;\Gamma_2;...;\langle \mathtt{t}_\infty, \mathtt{k}_\infty, \Gamma_\infty, \mathtt{v}_\infty \rangle$$

is not defined.

6.2. Basic statements

We consider the following basic statements: *assignment statement*, *procedure call*, *dynamic variable creation*, *return statement*, and *goto statement*.

Assignment statements

If e_l is a Σ-expression of type $loc(t)$ and e_r a Σ-expression either of type t or type $loc(t)$, then $e_l := e_r$ and $e_l = null$ are Σ-statements called the *assignment statements*, where the second statement is valid only if t is a location type (i.e., e_l is a pointer).

We first define the semantics of the statement for the case where t is not an array type and assume that the evaluation of both parts of the statement may change the state.

Semantics. Let $[\![e_l]\!]^{\mathtt{A(B)}} = \langle \Gamma_\mathtt{l}, \mathtt{l} \rangle$, $\mathtt{A}' = \mathtt{A}\Gamma_\mathtt{l}$, and $[\![e_r]\!]^{\mathtt{A}'\mathtt{(B)}} = \langle \Gamma_\mathtt{r}, \mathtt{v}_\mathtt{r} \rangle$. Then

$$[\![e_l.x = e_r]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, (\Gamma_\mathtt{l};\Gamma_\mathtt{r};\Gamma_\mathtt{a}), \perp \rangle,$$

where $\Gamma_\mathtt{a} = \{(\mathtt{loc(t)}, \mathtt{l}, \mathtt{v}_\mathtt{r})\}$ if e_r has type t, and $\Gamma_\mathtt{a} = \{(\mathtt{loc(t)}, \mathtt{l}, \mathtt{v}_\mathtt{r} \uparrow_{\mathtt{loc(t)},\mathtt{t}})\}$ if e_r has type $loc(t)$;

$$[\![e_l = null]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, (\Gamma_\mathtt{l};\Gamma_\mathtt{a}), \perp \rangle,$$

where $\Gamma_\mathtt{a} = \{(\mathtt{loc(t)}, \mathtt{v}_\mathtt{l}, \mathtt{nil})\}$.

The interpretation of the expression is not defined if $\mathtt{v_l} = \mathtt{nil}$ or $\mathtt{v_r}$ is not defined.

If $ar1$ and $ar2$ are array variables with n elements of type t, then the interpretation of the assignment $ar1 := ar2$ produces the update set

$$\Gamma_{\mathtt{a}} = \{(\mathtt{loc(t)}, \mathtt{l_1}, \mathtt{v_1}), ..., (\mathtt{loc(t)}, \mathtt{l_n}, \mathtt{v_n})\},$$

where $l_i = ar1 \uparrow (i)$ and $v_i = ar2 \uparrow (i) \uparrow$. In this way, all element locations of the first array are updated by the content of the corresponding element locations of the second array.

Procedure calls

If $(p, \langle u_1, ...u_n\rangle, \mathtt{void})$ is a procedure declaration, where u_i, $i = 1, ..., n$, is either t_i or *ref* t_i, and

$$e_i \text{ is } \begin{cases} \text{an expression of type } loc(t_i) \text{ when } u_i \text{ is } ref\, t_i, \\ \text{an expression either of type } t_i \text{ or } loc(t_i) \text{ in the opposite case,} \end{cases}$$

then $p(e_1, ..., e_n)$ is a Σ-statement called the *procedure call*.

Semantics. The semantics of the procedure call is the same as of the function call with the exception that a quadruple $\langle \mathtt{void}, \mathbf{O}, \Gamma, \bot\rangle$ is produced.

Dynamic variable creation

If p is a variable of type $loc(t)$, then $new(p)$ is a statement called the *dynamic variable creation*.

Semantics. $[\![new(p)]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, \Gamma, \bot\rangle$, where $\Gamma = \{\delta; \alpha\}$, $\delta = (\mathtt{loc(t)}, \mathtt{l})$, $\mathtt{l} \in \mathtt{B_{Locs}}$ and $\mathtt{l} \notin \mathtt{A_{loc}}$, $\mathtt{A'} = \mathtt{A}\delta$, and $\alpha = (\mathtt{loc(loc(t))}, \mathtt{p}^{\mathtt{A'}}, \mathtt{l})$.

In this way, the location sort $loc(t)$ is extended with a new element, and this element is assigned to p. If t is an array type $array(ind, t')$, then a set $\mathtt{L}$ of elements of type $loc(t')$ and an injective finite mapping $\mathtt{B_{ind}} \to \mathtt{L}$ are additionally created and $\mathtt{l}$ is initialized with this mapping.

Return statements

If e is a Σ-expression of type t, then **return** and **return** e are Σ-statements called the *return statements*.

Semantics. Let $[\![e]\!]^{\mathtt{A(B)}} = \langle \Gamma, \mathtt{v} \rangle$, then $[\![\textbf{return}]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{R}, \emptyset, \bot \rangle$ and $[\![\textbf{return } e]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathbf{R}, \Gamma, \mathtt{v} \rangle$.

Goto statements

If M is a label, then **goto** M is a statement called the *goto statement*.

Semantics. $[\![\textbf{goto } M]\!]^{\mathtt{A(B)}} = \langle \mathtt{string}, \mathbf{B}, \emptyset, \mathtt{M} \rangle$.

6.3. Statement constructors

Complex statements are constructed recursively from basic statements by means of several statement constructors. For convenience, we consider separately the sequence of statements not containing local declarations and the block statement containing declarations of local variables.

Sequences of statements

If $St_1, St_2, \ldots, St_n$ are Σ-statements, then $\{St_1; St_2; \ldots; St_n\}$ is a statement called the *sequence of* Σ*-statements*. For a sequence of statements $StSeq$, the function $\mathtt{length}(StSeq)$ produces the number of component statements, the set $\mathtt{labels}(StSeq)$ contains the labels of the statements $\{\ St_1; St_2; \ldots; St_n\ \}$ (empty set if none of the statements has a label), and the final mapping $\mathtt{lmap}^{\mathtt{StSeq}} : labels(StSeq) \to Nat$ associate with each label the sequential number of the corresponding statement.

Semantics. The semantics of a sequence of statements

$$StSeq = \{\ St_1; St_2; \ldots; St_n\ \}$$

is defined with the use of two parameters, the state $\mathtt{A(B)}$ and the sequential number i of the fist statement in $StSeq$ to be executed and denoted by $[\![StSeq^i]\!]^{\mathtt{A(B)}}$. Since the sequence of statements starts execution from the statement with the sequential number 1, we define: $[\![StSeq]\!]^{\mathtt{A(B)}} = [\![StSeq^1]\!]^{\mathtt{A(B)}}$.

Now, let $[\![St_i]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma_{\mathtt{i}}, \mathtt{v} \rangle$. If $\mathtt{k}$ is $\mathbf{R}$ (return statement has been executed), then $[\![StSeq^i]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma_{\mathtt{i}}, \mathbf{R} \rangle$ (exit the sequence by return statement). Otherwise:

If $\mathtt{k} = \mathbf{B}$ (goto statement has been executed)
then if $\mathtt{v} \in \mathtt{labels}(StSeq)$ (transfer of control to a statement of this sequence)
then let $\mathtt{j} = \mathtt{lmap}^{\mathtt{StSeq}}(\mathtt{v})$ and $\mathtt{A}' = \mathtt{A}\Gamma_\mathtt{i}$ in $[\![StSeq^i]\!]^{\mathtt{A(B)}} = \Gamma_\mathtt{i};[\![StSeq^j]\!]^{\mathtt{A}'\mathtt{(B)}}$
else $[\![StSeq^i]\!]^{\mathtt{A(B)}} = \langle \mathtt{string}, \mathbf{B}, \Gamma_\mathtt{i}, \mathtt{v} \rangle$ (exit the sequence by goto statement).

If $\mathtt{k} = \mathbf{O}$ (an ordinary statement has been executed)
then if $\mathtt{i} < \mathtt{length}(StSeq)$ (there are more statements in the sequence)
then let $\mathtt{j} = \mathtt{i} + \mathtt{1}$ and $\mathtt{A}' = \mathtt{A}\Gamma_\mathtt{i}$ in $[\![StSeq^i]\!]^{\mathtt{A(B)}} = \Gamma_\mathtt{i};[\![StSeq^j]\!]^{\mathtt{A}'\mathtt{(B)}}$ (proceed with the next statement)
else $[\![StSeq^i]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, \Gamma, \mathtt{v} \rangle$ (the sequence of statements is executed).

Thus, to execute a sequence of statements starting with a state $\mathtt{A}$, it is sufficient to create the sequential union of their update sets and use it for the transformation of $\mathtt{A}$ (which is equivalent to the sequential execution of the statements one after another). Note that the creation of the update set stops as soon as control is transferred outside of the sequence of statements.

The semantics of a sequence of statements is undefined if the semantics of one of the component statements is undefined or the infinite sequence of update sets is produced.

Blocks

For brevity, we consider that only non-initialized local variables can be declared in a block, any array type and location type used in local declarations belong to AT and LT, respectively, and declarations precede statements.

First of all, we introduce a notion of signature increment. Let $t_1, ..., t_n$ be types and $X = \{x_1, ..., x_n\}$ a set of identifiers, then the set

$$\Delta = \{(x_1, t_1), \ldots, (x_n, t_n)\}$$

is a *signature increment*.

The sequential union of a signature $\Sigma = (S, F)$ and a signature increment $\Delta = \{(x_1, t_1), ..., (x_n, t_n)\}$, denoted by $\Sigma;\Delta$, is defined as follows: $\Sigma;\Delta = (S, F_1)$, where $F_1 = \Delta \cup (F \backslash \{(x, t)\})$ for any $(x, t) \in F$ if there is an $(x, t') \in \Delta$. In this way any variable declared in a signature increment hides the previously declared variable or constant with the same identifier.

Let $\Sigma_0 = (S_0, F_0)$ be a program signature and Δ_1 a signature increment, then

$$\Sigma_1 = (S_0, F_1) = (\Sigma_0 ; \Delta_1)$$

is a *block signature*. Now, if $\Sigma_{n-1} = (\Sigma_{n-2} ; \Delta_{n-1})$ is a block signature and Δ_n is a signature increment, then $\Sigma_n = (\Sigma_{n-1} ; \Delta_n)$ is a *block signature*. Thus, a block signature generally looks as follows:

$$((((\Sigma_0 ; \Delta_1) ; ...) ; \Delta_{n-1}) ; \Delta_n).$$

In the sequel, we denote a block signature by $\Sigma' = (\Sigma ; \Delta_n)$, where

$$\Sigma = (((\Sigma_0 ; \Delta_1) ; ...) ; \Delta_{n-1}).$$

We also denote $\Delta_1 ; ... ; \Delta_n$ by Δ.

Now, if $\Sigma' = (\Sigma ; \Delta_n)$ is a block signature and $\mathtt{A}$ is a Σ-algebra, then a Σ'-algebra $\mathtt{A'}$ is constructed by extending $\mathtt{A}$ with a constant $\mathtt{x}^{\mathtt{A'}}$ of type $loc(t)$ for each $(x, t) \in \Delta_n$ (a finite mapping is also associated with $\mathtt{x}^{\mathtt{A'}}$ if x is an array variable, as described in Section 3.2). Note that if Σ is the program signature Σ_0, then a state algebra $\mathtt{A_0}$ is extended.

The rules for the construction of expressions using local variables are quite simple, for example:

– if (x, t) is a local variable from Δ, then x is a Σ'-expression of type t.
Interpretation: if $\mathtt{A'}$ is a Σ'-algebra, then $[\![x]\!]^{\mathtt{A'(B)}} = \mathtt{x}^{\mathtt{A'}}$.

Now, if $\Delta_n = \{(x_1, t_1), ..., (x_k, t_k)\}$ is a signature increment, $\Sigma' = (\Sigma ; \Delta_n)$ and $St_1, ..., St_m$ a sequence of Σ'-statements, then

$$\textbf{begin } \Delta_n ;\ St_1 ;\ St_2 ; \ldots ; St_m \textbf{ end}$$

is a Σ-statement called the *block*.

Semantics. Let $\mathtt{A'}$ be a Σ'-algebra and

$$[\![\{\ St_1 ;\ St_2 ; \ldots ; St_m\ \}]\!]^{\mathtt{A'(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma', \mathtt{v} \rangle$$

if $[\![\{\ St_1 ;\ St_2 ; \ldots ; St_m\ \}]\!]^{\mathtt{A'(B)}}$ is defined. Then

$$[\![\textbf{begin } \Delta_n ;\ St_1 ;\ St_2 ; \ldots ; St_m \textbf{ end}]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle,$$

where Γ is obtained from Γ' by deleting any location update $\alpha = (\mathtt{loc(t)}, \mathtt{x}^{\mathtt{A'}}, \mathtt{a})$ if x is a variable of type t from Δ_n. In this way all updates of local variables of the block are ignored.

If Δ_n contains an initialized local variable declaration $x : t = e$, then the algebra $\mathtt{A'}$ is updated by the the evaluation of the statement $x := e$.

If statements

If g is a Σ-expression of type Boolean and St and St_e are Σ-statements, then

$$\textbf{if } g \textbf{ then } St \text{ and } \textbf{ if } g \textbf{ then } St \textbf{ else } St_e$$

are Σ-statements called the *if statements*.

Semantics. Let $[\![g]\!]^{\mathtt{A(B)}} = \langle \Gamma_{\mathtt{g}}, \mathtt{g} \rangle$ and $\mathtt{A}' = \mathtt{A}\Gamma_{\mathtt{g}}$. Now,
if `g = true`
then if $[\![St]\!]^{\mathtt{A}'\mathtt{(B)}}$ is defined
then $[\![\textbf{if } g \textbf{ then } St]\!]^{\mathtt{A(B)}} = \Gamma_{\mathtt{g}};[\![St]\!]^{\mathtt{A}'\mathtt{(B)}}$ and
$[\![\textbf{if } g \textbf{ then } St \textbf{ else } St_e]\!]^{\mathtt{A(B)}} = \Gamma_{\mathtt{g}};[\![St]\!]^{\mathtt{A}'\mathtt{(B)}}$
else $[\![\textbf{if } g \textbf{ then } St]\!]^{\mathtt{A(B)}}$ and $[\![\textbf{if } g \textbf{ then } St \textbf{ else } St_e]\!]^{\mathtt{A(B)}}$ are undefined
else $[\![\textbf{if } g \textbf{ then } St]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, \Gamma_{\mathtt{g}}, \bot \rangle$ and
if $[\![St_e]\!]^{\mathtt{A}'\mathtt{(B)}}$ is defined,
then $[\![\textbf{if } g \textbf{ then } St \textbf{ else } St_e]\!]^{\mathtt{A(B)}} = \Gamma_{\mathtt{g}};[\![St_e]\!]^{\mathtt{A}'\mathtt{(B)}}$
else $[\![\textbf{if } g \textbf{ then } St \textbf{ else } St_e]\!]^{\mathtt{A(B)}}$ is undefined.

Case statements

If e is a Σ-expression of a scalar type t, CL_{11}, ..., CL_{1k_1}, ... , CL_{m1},..., CL_{mk_m} are different Σ_{dat}-expressions (constants) called *case labels*, and St_1, ..., St_m are Σ-statements, then

case e **of**
$CL_{11},\ ...,\ CL_{1k_1} : St_1;$
$\vdots$
$CL_{m1},\ ...,\ CL_{mk_m} : St_m$
end is a Σ-statement called the *case statement*.

Semantics. Let $St =$
case e **of**
$CL_{11},\ ...,\ CL_{1k_1} : St_1;$
$\vdots$
$CL_{m1},\ ...,\ CL_{mk_m} : St_m$
end

Since any case label is a constant expression, all labels can be evaluated in the same state $\mathtt{A}$, and the evaluation produces the empty update set. Let $[\![e]\!]^{\mathtt{A(B)}} = \langle \Gamma_{\mathtt{e}}, \mathtt{v_e} \rangle$, $[\![CL_{ij}]\!]^{\mathtt{A(B)}} = \langle \emptyset, \mathtt{v_{ij}} \rangle$, and $\mathtt{A}' = \mathtt{A}\Gamma_{\mathtt{e}}$. Now, if CL_{ij} is the

first label in the sequence $CL_{11}, ..., CL_{1k_1}, ..., CL_{m1}, ..., CL_{mk_m}$ such that $\mathtt{v_e} = \mathtt{v_{ij}}$, then $[\![St]\!]^{\mathtt{A}'(\mathtt{B})} = [\![St_i]\!]^{\mathtt{A}'(\mathtt{B})}$ else $[\![St]\!]^{\mathtt{A}'(\mathtt{B})} = \langle \mathtt{void}, \mathbf{O}, \emptyset, \bot \rangle$.

The semantics of St is not defined if $[\![St_i]\!]^{\mathtt{A}'(\mathtt{B})}$ is not defined.

While loops

If St is a Σ-statement and g a Σ-expression of type Boolean, then **while** g **do** St is a Σ-statement called the *while loop*.

Semantics. Let $[\![g]\!]^{\mathtt{A}(\mathtt{B})} = \langle \Gamma_{\mathtt{g}}, \mathtt{g} \rangle$ and $\mathtt{A}' = \mathtt{A}\Gamma_{\mathtt{g}}$. Now,
if $\mathtt{g} = \mathtt{false}$
then $[\![\textbf{while } g \textbf{ do } St]\!]^{\mathtt{A}(\mathtt{B})} = \langle \mathtt{void}, \mathbf{O}, \Gamma_{\mathtt{g}}, \bot \rangle$
else let $\mathtt{A}' = \mathtt{A}\Gamma_{\mathtt{g}}$, $[\![St]\!]^{\mathtt{A}'(\mathtt{B})} = \langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$ and $\mathtt{A}'' = \mathtt{A}'\Gamma$ in
  if $\mathtt{k} = \mathbf{O}$ (St completes normally)
  then $[\![\textbf{while } g \textbf{ do } St]\!]^{\mathtt{A}(\mathtt{B})} = \Gamma_{\mathtt{g}};\Gamma;[\![\textbf{while } g \textbf{ do } St]\!]^{\mathtt{A}''(\mathtt{B})}$
  else (St completes abnormally)
    $[\![\textbf{while } g \textbf{ do } St]\!]^{\mathtt{A}(\mathtt{B})} = \langle \mathtt{t}, \mathtt{k}, (\Gamma_{\mathtt{g}};\Gamma), \mathtt{v} \rangle$.

Note that $[\![\textbf{while } g \textbf{ do } St]\!]^{\mathtt{A}(\mathtt{B})}$ is undefined if the semantics of St is undefined at some level of iteration or the infinite sequence of sequential union operations is produced.

Repeat loops

If St is a Σ-statement and g is a Σ-expression of type Boolean, then **repeat** St **until** g is a Σ-statement called the *repeat loop*.

Semantics. Let $[\![St]\!]^{\mathtt{A}(\mathtt{B})} = \langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$ and $\mathtt{A}' = \mathtt{A}\Gamma$ if $[\![St]\!]^{\mathtt{A}(\mathtt{B})}$ is defined. Now,
  if $\mathtt{k} = \mathbf{O}$ (St completes normally)
  then let $[\![g]\!]^{\mathtt{A}'(\mathtt{B})} = \langle \Gamma_{\mathtt{g}}, \mathtt{g} \rangle$ and $\mathtt{A}'' = \mathtt{A}'\Gamma_{\mathtt{g}}$ in
    if $\mathtt{g} = \mathtt{false}$ (iteration should be continued)
    then $[\![\textbf{repeat } St \textbf{ until } g]\!]^{\mathtt{A}(\mathtt{B})} = \Gamma;\Gamma_{\mathtt{g}};\ [\![\textbf{repeat } St \textbf{ until } g]\!]^{\mathtt{A}'(\mathtt{B})}$
    else $[\![\textbf{repeat } St \textbf{ until } g]\!]^{\mathtt{A}(\mathtt{B})} = \langle \mathtt{void}, \mathbf{O}, (\Gamma;\Gamma_{\mathtt{g}}), \bot \rangle$ (finish normally)
  else (St completes abnormally)
    $[\![\textbf{repeat } St \textbf{ until } g]\!]^{\mathtt{A}(\mathtt{B})} = \langle \mathtt{t}, \mathtt{k}, (\Gamma;\Gamma_{\mathtt{g}}), \mathtt{v} \rangle$.

$[\![\textbf{repeat } St \textbf{ until } g]\!]^{\mathtt{A}(\mathtt{B})}$ is undefined if the semantics of St is undefined at some level of iteration or an infinite sequence of sequential union operations is produced.

For loops

If i is a variable of a scalar type t (i.e., of type possessing the operation $succ : t \to t$), e_1 and e_2 are Σ-expressions of type t such that $[\![e_1]\!]^{\mathtt{A(B)}} = \langle \emptyset, \mathtt{v}_1 \rangle$ and $[\![e_2]\!]^{\mathtt{A(B)}} = \langle \emptyset, \mathtt{v}_2 \rangle$, and St is a Σ-statement, then

$$\textbf{for } i \ := \ e_1 \textbf{ to } e_2 \textbf{ do } St$$

is a Σ-statement called the *for loop*.

Semantics. Let i be an algebraic variable and τ_1 and τ_2 ground terms (constant expression) of type t evaluating in $\mathtt{A}$ to $\mathtt{v}_1$ and $\mathtt{v}_2$, respectively. Then, according to the specification of Pascal, the above statement is equivalent to the following one: **for** $i \ = \ \tau_1$ **to** τ_2 **do** St.
This means that both i and τ_1 and τ_2 cannot be updated by St, and the value of i is undefined when the loop is exited. In the sequel, the notation $St[\tau_1/i]$ means the substitution of each i in St by τ_1. The semantics of

$$\textbf{for } i \ = \ \tau_1 \textbf{ to } \tau_2 \textbf{ do } St$$

can be defined as follows:

Let $[\![St[\tau_1/i]]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$ and $\mathtt{A}' = \mathtt{A}\Gamma$ if $[\![St[\tau_1/i]]\!]^{\mathtt{A(B)}}$ is defined. Now,
  if $\mathtt{k} = \mathbf{O}$ (St completes normally)
  then if $\tau_1 \leq \tau_2$ (iteration should be continued)
    then $[\![\textbf{for } i \ = \ \tau_1 \textbf{ to } \tau_2 \textbf{ do } St]\!]^{\mathtt{A(B)}} = \Gamma ; [\![\textbf{for } i \ = \ succ(\tau_1) \textbf{ to } \tau_2 \textbf{ do } St]\!]^{\mathtt{A}'(\mathtt{B})}$
    else $[\![\textbf{for } i \ = \ \tau_1 \textbf{ to } \tau_2 \textbf{ do } St]\!]^{\mathtt{A(B)}} = \langle \mathtt{void}, \mathbf{O}, \emptyset, \perp \rangle$ (finish)
  else (St completes abnormally)
    $[\![\textbf{for } i \ = \ \tau_1 \textbf{ to } \tau_2 \textbf{ do } St]\!]^{\mathtt{A(B)}} = \langle \mathtt{t}, \mathtt{k}, \Gamma, \mathtt{v} \rangle$ (exit).

The semantics of **for** $i \ = \ \tau_1$ **to** τ_2 **do** St is undefined if the semantics of St is undefined at some level of iteration or an infinite sequence of sequential union operations is produced.

7. PROGRAM SPECIFICATION

A program specification extends a program signature by the definition of each function and each constant in the signature.

Let f_{rt} be a function symbol in Σ_0, where $r = u_1 ... u_n$ such that u_i is either t_i or *ref* t_i, $\Delta = \{(p_1, t_1), ..., (p_n, s_n)\}$ is a signature increment, and St a Σ_0-block with the signature increment Δ, then

$$f(p_1 : u_1, ..., p_n : u_n) : t;\ St;$$

is a *function definition.*

A *program specification* $\mathcal{PS}$ for a program schema $\mathcal{S}$ is a set of function definitions for all functions declared in $\mathcal{S}$.

Semantics. Let $f(e_1, ..., e_n)$ be a function call in the signature Σ (it may be the signature of some block), where $e_1, ..., e_n$ are Σ-expressions as described in Section 5, item 3. Let also $\mathtt{A}$ be a Σ-algebra, $\mathtt{v_1}, ..., \mathtt{v_n}$ results of the evaluation of $e_1, ..., e_n$, $\mathtt{A_n}$ a Σ-algebra resulting after the evaluation of e_n as described in Section 5, item 3, $\mathtt{A_0} = \mathtt{A_n}|_{\Sigma_0}$, and $\mathtt{A'}$ a (Σ, Δ)-algebra extending $\mathtt{A_0}$ in the following way: $\mathtt{p_i^{A'}}\uparrow = \mathtt{v_i}$ if u_i is t_i and $\mathtt{p_i^{A'}} = \mathtt{v_i}$ if u_i is $ref\ t_i$, $i = 1, ..., n$. Note that in the first case p_i is a local variable initialized by the value of the actual parameter, and in the second case p_i is a local constant equal to the actual parameter (both $\mathtt{p_i^{A'}}$ and $\mathtt{v_i}$ are the same location). Note also that, using the reduct $\mathtt{A_n}|_{\Sigma_0}$, we have obtained an algebra of the signature where the block St is constructed (i.e. an algebra of the signature (Σ_0, Δ)). This algebra extends the state algebra existing at the moment of the function invocation by the values of the actual parameters. The reduct is safe under our restriction that only local variables can be declared in a block (this means that all sorts in the reduct are the same as in the original algebra). Now, let

$$[\![f(e_1, ..., e_n)]\!]^{\mathtt{A(B)}} = \langle \Gamma, \mathtt{v} \rangle$$

and $[\![St]\!]^{\mathtt{A'(B)}} = \langle \mathtt{t_{st}}, \mathtt{k_{st}}, \Gamma_{\mathtt{st}}, \mathtt{v_{st}} \rangle$.
If the result type t is `void`, then the function definition

$$f(p_1 : u_1, ..., p_n : u_n) : t;\ St;$$

is *well-formed* iff $\mathtt{k_{st}}$ is either $\mathbf{O}$ or $\mathbf{R}$ and $\mathtt{t_{st}}$ is `void` (no return statement or no expression in the return statement). If the result type t is not `void`, the function definition $f(p_1 : u_1, ..., p_n : u_n) : t;\ St;$ is *well-formed* iff k_{st} is $\mathbf{R}$ and $\mathtt{t_{st}} = \mathtt{t}$ (the result of the expression in the return statement is of type t).

A program $\mathtt{P(B)}$ satisfies the function definition

$$f(p_1 : u_1, ..., p_n : u_n) : t;\ St;$$

in the program specification $\mathcal{PS}$ iff for any Σ where the call of f is constructed, for any Σ-state $\mathtt{A}$ and any Σ-expressions $e_1, ..., e_n$ it holds:

$$\mathtt{A}\Gamma = \mathtt{A}\Gamma_{\mathtt{st}} \text{ and } \mathtt{v} = \mathtt{v}_{\mathtt{st}}.$$

Note that we require the equivalence of the resulting states rather than equivalence of the resulting update sets because update sets may be different due to some optimizations of the function body.

In a similar way, if $(c, t) \in const$ is a constant declaration and e is a Σ_{dat}-expression (a constant expression) of type t, then $c : t = e$ is a *constant definition*. A program $\mathtt{P}(\mathtt{B})$ satisfies the constant definition $c : t = e$ iff for any Σ-state $\mathtt{A}$ it holds: $\mathtt{c}^{\mathtt{A}} = \mathtt{e}^{\mathtt{B}}$.

A program `P(B)` satisfies the program specification $\mathcal{PS}$ iff it satisfies each function definition and each constant definition in $\mathcal{PS}$.

8. CONCLUSION

We have introduced a formal model of an imperative program in the style of Abstract State Machines and have shown how some typical expressions and statements can be constructed and interpreted in terms of this model. The novelty of the approach consists in defining a specialized abstract state-based model whose components naturally represent program components. This makes our approach substantially different from traditional algebraic approaches where the program behavior is described in terms of axioms or rewriting rules [3] and the state is represented as a complex data type with a large number of auxiliary operations [4, 17, 18].

A clear way of defining the semantics of an imperative language is thus provided: basing on the type system of the language, define a program signature and its model, then give the interpretations of expressions and statements in terms of this model and use them for function specifications. A simple Pascal-like language has been chosen in this work to illustrate the approach. However, it can be easily adapted to a more complex language, and our experience with formalization of Java has proved this [10]. A combination of the location model described in this paper and object model of [10] can be used for defining semantics of C# [11]. This remains a subject of further work.

The author thanks Anureev I.S. for valuable comments on the first version of the paper.

REFERENCES

1. C. Calcagno, S. Ishtiaq, and P.W. O'Hearn. Semantic Analysis of Pointer Aliasing, Allocation and Disposal in Hoare Logic. *Proc. 2nd Intern. Conf. on Principles and Practice of Declarative Programming*, 2000.
2. J. Cosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification (Second Edition)*. Addison-Wesley, 2000.
3. A. van Deursen, J. Heering, and P. Klint, eds. Language Prototyping: An Algebraic Specification Approach. *AMAST Series in Computing*, vol. 5, World Scientific, 1996.
4. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*, The MIT Press, 1996.
5. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, Oxford University Press, 1995, pp. 9-36.
6. Y. Gurevich and J. Huggins. The semantics of the C programming language. *Computer Science Logic*, LNCS, vol. 702, 1993, pp. 274-309.
7. Y. Gurevich, *May 1997 Draft of the ASM Guide*. Available electronically from http://www.eecs.umich.edu/gasm/.
8. Robert Eschbach, U. Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines. *Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, eds., Abstract State Machines: Theory and Applications*, LNCS, vol. 1912, 2000, pp. 242-265.
9. Philipp W. Kutter and Alfonso Pierantonio. The Formal Specification of Oberon, *Journal of Universal Computer Science*, 1997, vol. 3, no. 5, pp. 443–503.
10. K. Lellahi, A. Zamulin. *Implicit State Approach for Formalization of Sequential Java-like Programs*. Technical report of LIPN, Univ. Paris 13, 2002.
11. Microsoft Corp., et al. `C#` language specification.Drafts of the ECMA TC39/TG3 standartization process. `http://msdn.microsoft.com/net/ecma/`, 2001.
12. B. Möller. Towards Pointer Algebra. *Science of Computer Programming*, vol. 21, 1993, pp. 57-90.
13. P.D. Mosses. The Varieties of Programming Language Semantics (And Their Uses). *Perspectives in System Informatics*, LNCS, vol. 2244, pp. 165-190.
14. J. C. Reynolds. Separation Logic: A Logic fro Shared Mutable data Structures. *Proc. 7th Annual IEEE Symp. on Logic in Computer Science*, 2002.
15. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
16. Charles Wallace. The Semantics of the C++ Programming Language. *Specification and Validation Methods, ed. E. B$\ddot{\text{Y}}$rger*, Oxford University Press, 1995, pp. 131–164.
17. A.V. Zamulin. Algebraic Modelling of Imperative Languages With Pointers. *Formal Methods in Programming and Their Applications*, LNCS, vol. 735, 1993, pp. 81 - 97.
18. A.V. Zamulin. Algebraic Semantics of Imperative Statements. *Australian Computer Science Communications*, vol. 17, No 1, 1994, pp. 581-587.
19. Wolf Zimmerman, and Thilo Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach, *Journal of Universal Computer Science*, 1997, vol. 3, no. 5, pp. 504–567.

А. В. Замулин

ФОРМАЛЬНАЯ МОДЕЛЬ ПАСКАЛЕПОДОБНОЙ ПРОГРАММЫ

Препринт
104

Рукопись поступила в редакцию 20.01.2003
Рецензент И. С. Ануреев
Редактор Н. А. Черемных

Подписано в печать 15.03.2003
Формат бумаги 60×84 1/16 Объем 1,5 уч.-изд.л., 1,6 п.л.
Тираж 50 экз.

НФ ООО ИПО "Эмари" РИЦ, 630090, г. Новосибирск, пр. Акад. Лаврентьева, 6