Joint NCC & IIS Bull., Comp. Science, 20 (2004), 113–128

Algebraic semantics of an imperative programming language as a compiler abstract model*

A. V. Zamulin

Abstract. It is shown in the paper that state-based algebraic semantics of an imperative program can be regarded as a compiler abstract model and can serve as a good assistant of the compiler designer.

1. Introduction

A new mechanism of definition of the semantics of an imperative programming language is proposed in [4]. The mechanism is based on the construction of a mathematical model of a program, so that model components are used as denotations of language constructions. This approach follows the style of algebraic specifications [1] where first a signature, then a set of its models, then construction and interpretation of the terms of this signature, and finally what is a specification and what is the set of its models are defined. According to this, it is defined in [4]:

- how the signature of a Pascal-like program containing the symbols to be used in expressions and statements is constructed;
- what is the set of models (arbitrary object programs) of this signature;
- how terms (expressions and statements of this language) are constructed and interpreted;
- what is a program specification and what is the set of its models (valid object programs).

The program state is represented in this model by a many-sorted algebra whose components directly match the components of the state of a real program, and program functions and procedures are represented by mathematical functions analyzing and/or changing the program state. In this way, the abstract program model consists only of those components that have direct counterparts in a real program.

The semantics of expressions and statements of a programming language is defined in terms of the components of the abstract program model. Since

*The work is supported in part by Russian Foundation for Basic Research under Grant N 04-01-00272.

the components are high-level entities (mainly functions), the semantics is natural and easy to understand.

In this paper, we want to show that this kind of semantics of a programming language is also an abstract model of a compiler, and it can serve as an effective assistant of compiler designers.

2. Program signature and compiler symbol table

2.1. Basic signature

We illustrate the approach basing on a simple type system with the following grammar:

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

where `BASE` is a set of primitive types (for instance, `Boolean`, `integer`, `real`, `from_one_to_ten` and `char`), `array` is a Pascal-like array type constructor, and `loc` is a location (pointer) type constructor. A special type `void` not belonging to $\mathcal{T}$ will be used as a result type of a function producing nothing except a new program state.

Thus, our type system contains the part `BASE` with the basic signature $\Sigma_{dat} = (S_{dat}, F_{dat})$, where S_{dat} is the set of primitive data types and F_{dat} is the set of their operation symbols (names or operators) indexed by their functionalities (i.e., by the types of their arguments and results; an operation without arguments is called a *constant*).

Given a programming language, the basic signature is the basic part of the compiler symbol table in the process of compilation of any program of this language. In our case, for instance, the symbol table will contain the names `Boolean`, `integer`, `real`, `from_one_to_ten` and `char` as the names of primitive types, the names `array` and `loc` as the names of type constructors, and the symbols of all operations of primitive data types with indication of their functionalities.

2.2. Program schema

The program schema defines the symbols that may be used in the program in addition to the symbols of the basic signature. It is constructed on the base of variable, constant and function declarations. Formally, the *program schema* $\mathcal{S}$ over the type system $\mathcal{T}$ consists of:

1) a finite set of array types AT,

2) a finite set of location types LT,

3) three finite mappings var : `IDENT` $\to \mathcal{T}$, $const$: `IDENT` $\to$ `BASE`, and $func$: `IDENT` $\times \mathcal{T}^* \longrightarrow \mathcal{T}^v$, where `IDENT` is a nonempty set of *identifiers*, $\mathcal{T}^*$ the set of strings of elements each being either t or $ref\ t$, (t is a type from the set $\mathcal{T}$), and $\mathcal{T}^v$ the set $\mathcal{T} \cup \{\texttt{void}\}$. 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 in the end of the paper.

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* (in a real program, the result type `void` in the procedure declaration may be omitted).

Example. The schema of the following fragment of a Pascal program:

```
const c: integer, d: char;
var x: real, q: ↑integer,
    m: array from_one_to_ten of Boolean;
func f(integer, var Boolean): char;
proc p(real);
```

is as follows:

AT = {`array(from_one_to_ten, Boolean)`};
LT = {`loc(integer)`};
$const$ = {`(c, integer), (d, char)`};
var = {`(x, real), (q, loc(integer)),`
`(m, array(from_one_to_ten, Boolean))`};
$func$ = {`(f, ⟨integer, ref Boolean⟩, char), (p, ⟨real⟩, void)`}.

2.3. Program and block signature

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

- $S = S_{dat} \cup AT \cup LT \cup \{loc(t) \mid \exists\ (x, t) \in var\}$;
- $F = F_{dat} \biguplus var \biguplus const \biguplus (func \cup F_{acc})$, where F_{acc} is a set containing the *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 the access function symbol is created for each location type.

Example. Denote the type `array(from_one_to_ten, Boolean)` by `AR`. Then, ignoring the operation symbols of the primitive types, we can compose the following signature for our program schema:

S = {`Boolean, integer, real, from_one_to_ten,`
`char, AR, loc(integer), loc(real),`
`loc(loc(integer)), loc(AR)`};
F = {`var'(x, real), var'(q, loc(integer)), var'(m, AR)),`
`const'(c, integer), const'(d, char),`
`func'(f, ⟨integer, ref Boolean⟩, char),`
`func'(p, real, void), func'(_↑, loc(integer), integer),`

$\texttt{func}'(\_\uparrow, \texttt{loc(real)}, \texttt{real})$,
$\texttt{func}'(\_\uparrow, \texttt{loc(loc(integer))}, \texttt{loc(integer)})$,
$\texttt{func}'(\_\uparrow, \texttt{loc(AR)}, \texttt{AR})\}$.

It is easy to notice that the program signature is a "template" of the compiler symbol table (the term "template" means here that constants are not yet provided with values, variables with addresses, etc.). Thus, the set S indicates which types are used in the program, and the set F indicates which symbols may be used in its expressions and statements. All symbols in F are partitioned into the groups corresponding to variable, constant, and function declarations. Each symbol is accompanied by its *functionality*. This is the type of a value for a constant, the type of a location for a variable, and the type of a function for a function symbol.

A set of symbols can also be declared in a block. Formally, this set is modelled by the notion of signature increment defined in the simplest case as follows. Let $t_1, ..., t_n$ be types and $X = \{x_1, ..., x_n\}$ a set of variable identifiers, then the set

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

is a *signature increment*.

Using a source signature $\Sigma = (S, F)$ and the signature increment Δ, we construct the block signature $\Sigma' = (S, F')$ as follows:

1) the signature Σ is reduced to the intermediate signature $\Sigma'' = (S, F'')$ by defining $F'' = F \backslash \{(x, t) \mid \exists t'.(x, t') \in \Delta\}$ (i.e., by deleting the names redeclared in the block from the source signature);

2) the signature $\Sigma' = (S, F')$ is extended to the signature $\Sigma' = (S, F')$ by defining $F' = F'' \cup \Delta$ (i.e., by adding the names declared in the block to the intermediate signature Σ'').

In this way, the *block signature* Σ' is constructed as an extension of the reduced source signature by new symbols. Note that, as a result, the block signature does not possess the symbols redeclared in the block, which corresponds to the principle of symbol hiding in the traditional block-structured programming languages.

Example. Let the following variables be declared in block Bl:

```
var x: integer, q: real.
```

This gives us the following signature increment:

$\Delta =$ `{(x, integer), (q, real)}`

which extends our signature Σ to the following block signature:

$S = \{$`Boolean, integer, real, from_one_to_ten,`
`char, AR, loc(integer), loc(real),`
`loc(loc(integer)), loc(AR)`$\}$;
$F' = \{\texttt{var}'(\texttt{x}, \texttt{integer}), \texttt{var}'(\texttt{q}, \texttt{real}), \texttt{var}'(\texttt{m}, \texttt{AR}),$

$\mathtt{const'(c, integer)}, \mathtt{const'(d, char)},$
$\mathtt{func'(f, \langle integer, ref\ Boolean\rangle, char)},$
$\mathtt{func'(p, real, void)}, \mathtt{func'(\_\uparrow, loc(integer), integer)},$
$\mathtt{func'(\_\uparrow, loc(real), real)},$
$\mathtt{func'(\_\uparrow, loc(loc(integer)), loc(integer))},$
$\mathtt{func'(\_\uparrow, loc(AR), AR)}\}.$

When we exit the block, we just return to the source signature.

To avoid storing a chain of signatures (the chain can be quite long in case of a deep block nesting), the symbol table in the compiler is partitioned into blocks corresponding to the program signature and signature increments, and identifier search starts in the deepest block. The return to the previous signature is done just by deleting the latest signature increment.

3. Program algebra and program state

The program state is formally represented by a many-sorted algebra whose components are the sets of data type values, primitive type operations, access functions, constant values, variable locations, etc. Two main parts can be distinguished in this algebra: the *static part* (basic algebra) mainly consisting of values and operations of the primitive types, and *dynamic part* mainly consisting of access functions and dynamically created memory locations.

3.1. Basic algebra

The *basic algebra* defines the sets and operations that are basic for the other program components. It is a formal counterpart of the implementations of the primitive types in a given compiler. A *basic algebra* $\mathtt{B}$ associates with each type t a set $\mathtt{B_t}$, its *carrier*, with each operation symbol of t a partial function, its *implementation*, and with each constant an element of the corresponding carrier. In addition to this, the components of the basic algebra are a special set $\mathtt{B_{Locs}}$ containing elements of all location types, a special element `nil` not belonging to any carrier, and element $\perp$ as a single value of type `void`. The carriers are pairwise disjoint.

In a particular compiler, the set $\mathtt{B_{Locs}}$ may be a set of addresses to be allocated to variables in the stack or heap, the element `nil` the null address, the set $\mathtt{B_{char}}$ a set of bytes, the set $\mathtt{B_{real}}$ a set of 8-byte elements, and so on. A function associated with an operation symbol may be one or more machine instructions implementing this operation (it is an inline function as a rule).

3.2. Dynamic part of the program state

The dynamic part of the program state extends the basic algebra $\mathtt{B}$ to a *state algebra* $\mathtt{A}$ in accordance with the extension of the basic signature Σ_{dat} to the program signature $\Sigma = (S, F)$. This extension, called the Σ-*state*, is created as follows:

- $\mathtt{A}_\tau = \mathtt{B}_\tau$ for any primitive type t (i.e., the carriers of all primitive types remain the same);
- 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' (i.e, the sets of elements of each location type in each program state have only one common element, $\mathtt{nil}$);
- a set $\mathtt{A}_{array(ind,t)}$ of finite injective mappings $\mathtt{A}_{ind} \to \mathtt{A}^\circ_{loc(t)}$ is associated with each array type $array(ind, t)$ (i.e., an array is a mapping of the index type to the element type such that each index value is mapped to a separate location);
- an element, $\mathtt{c}^\mathtt{A} \in \mathtt{A}_t$, is associated with each constant declaration (c, t) (i.e., a value of the indicated type is associated with each constant identifier);
- an element, $\mathtt{x}^\mathtt{A} \in \mathtt{A}^\circ_{loc(t)}$, called *location constant*, is associated with each variable declaration (x, t) (in contrast to the constant, a location of the indicated type is associated with a variable identifier);
- a partial function $\_!^\mathtt{A} : \mathtt{A}_{loc(t)} \to \mathtt{A}_t$ is associated with each access function symbol $\_!_{loc(i),t}$ (i.e, a variable location may be initialized with a value of the corresponding type).

In this way, each location type is provided with a set of locations (called a *location sort* in the sequel) and a special element $\mathtt{nil}$, each constant is provided with a value, and each variable is provided with a location that may be initialized.

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,
- each array location is initialized.

Thus, the notion of the well-formedness of the state algebra is a direct indication for the compiler designer that each program variable should be supplied with a separate address, and, in addition, each array variable should

be supplied with the needed number of element type locations (this means that an array itself and each of its elements are provided with a location). Note that the array element locations are not required to be adjacent in a well-formed state algebra. The placement of array elements in adjacent locations may be considered as a natural means of program optimization. It is interesting to note that although in a real object program the address of an array and the address of its first element are the same address, formally they are different addresses (locations) of different types.

A location $\mathtt{l}$ is said to be *allocated* in a state $\mathtt{A}$ if $\mathtt{l} \in \mathtt{A}_{\mathtt{loc(t)}}$ for some $loc(t) \in S$. The set of all allocated locations in $\mathtt{A}$ is denoted by $\mathtt{A}_{\mathtt{loc}}$ in the sequel.

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

As it is indicate above, when we enter a block with a signature increment Δ, we move from a source signature Σ to the block signature $\Sigma' = (S, F')$ that contains new symbols of location constants (variable identifiers). In this case, we need to transform some Σ-algebra $\mathtt{A}$ to a Σ'-algebra $\mathtt{A}'$ by allocating memory for local variables. Formally, this is performed in two steps:

1) the algebra $\mathtt{A}$ is defined to be a Σ''-algebra $\mathtt{A}''$ (in this way, the components redeclared in the block become invisible);

2) the algebra $\mathtt{A}''$ is extended with a constant $\mathtt{x}^{\mathtt{A}'}$ of type $loc(t)$ for each pair $(x, t) \in \Delta$ (a finite mapping is also associated with $\mathtt{x}^{\mathtt{A}'}$ if x is an array variable).

In this way, we have constructed a block algebra that does not possess the redeclared components and possesses a location for each local variable. The corresponding sets of locations $\mathtt{A}_{\mathtt{loc(t)}}$ and finite mappings $\mathtt{A}_{\mathtt{ind}} \to \mathtt{A}^{\circ}_{\mathtt{loc(t)}}$ are also extended in the new algebra (to take into account the newly allocated locations). The requirement of algebra correctness is also extended in an evident way.

When the block finishes its work, we get some Σ'-algebra $\mathtt{A}'_1$ in which some locations of the original algebra $\mathtt{A}$ may have a new content. To fix this new content and get rid of local block locations, we do the following actions:

1) the algebra $\mathtt{A}'_1$ is reduced to a Σ''-algebra $\mathtt{A}''_1$ by deleting the components associated with the elements of the set $F'\backslash F''$ (i.e., by deleting the components declared in the block);

2) the algebra $\mathtt{A}''_1$ is extended to a Σ-algebra $\mathtt{A}_1$ by adding the components of the algebra $\mathtt{A}$ associated with the elements of the set $F\backslash F''$ (i.e., the components redeclared in the block are actually restored).

The resulting algebra $\mathtt{A}_1$ possesses all updates of the original algebra. It is clear that this formalism of entering and exiting the block naturally models the program stack pushed in when the block is entered and popped

out when it is exited. Note that, in contrast to the traditional algebraic semantics (see [3]), we did not need to define an extra type "stack" with corresponding operations; everything is easily modelled with the use of the notions of algebra extension and reduction.

A location sort or access function may be different in different states, i.e., locations may be dynamically allocated and deleted in the process of program execution, and a variable may be assigned different values. However, different Σ-states must have the same basic algebra (i.e., the same data type implementation). We denote the set of all Σ-states with the same base $\mathtt{B}$ by $\mathtt{state_B}(\Sigma)$ and mean by a Σ_B-state a Σ-state with the basic algebra $\mathtt{B}$.

3.3. State update

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.

A location update *in a Σ_B-state $\mathtt{A}$ is a pair $(\mathtt{l}, \mathtt{a})$, where $\mathtt{l}$ is an element of the set $\mathtt{A_{loc(t)}}$ such that $\mathtt{t}$ is not an array type and $\mathtt{a}$ an element of the set $\mathtt{A_t}$.* ■

Transformation of a Σ_B-state $\mathtt{A}$ into a Σ_B-state $\mathtt{A}\alpha$ by the location update $\alpha = (\mathtt{l}, \mathtt{a})$ is performed by updating the access function $\_\uparrow_{loc(t),t}$ in such a way that $(\mathtt{l}\uparrow)^{\mathtt{A}\alpha} = \mathtt{a}$. It is evident that the content of only one location is changed by this update which corresponds to the assignment of a new value to a variable or the machine instruction of updating a memory location. Note that an array location may not be updated, i.e., it cannot be assigned a new finite mapping (otherwise, the locations of the array elements would be changed); however, the location of any array element may be updated.

A sort update δ *in a Σ_B-state $\mathtt{A}$ is a pair $(\mathtt{A_{loc(t)}}, \mathtt{l})$, where $\mathtt{l}$ is a location such that $\mathtt{l} \in \mathtt{B_{Locs}}$ and $\mathtt{l} \notin \mathtt{A_{loc}}$.* ■

Transformation of a Σ_B-state $\mathtt{A}$ into a Σ_B-state $\mathtt{A}\delta$ by the sort update $\delta = (\mathtt{A_{loc(t)}}, \mathtt{l})$ is performed by inserting the location $\mathtt{l}$ into the set $\mathtt{A_{loc(t)}}$. In this way, the operation of allocating a new memory location is modelled. The answer to the question of where the new location is stored will be given in the next section.

4. Program

The abstract program model is a counterpart of the object program possessing a set of states and a set of functions and procedures manipulating the state. Recall that the notion of function in programming languages is different from the notion of function in mathematics since the program

function may have a side effect (update of the program state). One can however convert the program function into the mathematical one by representing its result as a pair, *state* and *value*. Let $|\mathtt{P(B)}| \subset \mathtt{state_B}(\Sigma)$ be the set of the states of a program $\mathtt{P(B)}$ and t be a type. Then $|\mathtt{P(B)}|_\mathtt{t} = \{\langle \mathtt{A}, \mathtt{v}\rangle \mid \mathtt{A} \in |\mathtt{P(B)}|\ \ \mathtt{v} \in \mathtt{A_t}\}$ is a set of pairs with a state algebra as the first element and an element of the state algebra as the second element. This set may be used as the codomain of the functions declared in the program. In what follows, for any $\Sigma_\mathtt{B}$-state $\mathtt{A}$ and any sequence of types $r = t_1, \ldots, t_n$, we denote the Cartesian product $\mathtt{A_{t_1}} \times \cdots \times \mathtt{A_{t_n}}$ by $\mathtt{A_r}$, which is a singleton set if $n = 0$.

Program $\mathtt{P(B)}$ of 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*), and
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

$$\mathtt{f_r^A} : \mathtt{A_{u_1}} \times \ldots \times \mathtt{A_{u_n}} \to |\mathtt{P(B)}|_\mathtt{t},$$

where $\mathtt{A_{u_i}}$ is $\mathtt{A^\circ_{loc(t_i)}}$ if u_i is $\mathit{ref}\ t_i$, and $\mathtt{A_{t_i}}$ 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 $\mathtt{A^\circ_{loc(t)}}$ 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). In other words, the program state is an implicit function parameter. A function always produces a pair, an algebra and value (an element of this algebra).

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. Function call

It is shown in the previous section that the set $\mathtt{A_{loc(t)}}$ is used as a function domain component in algebra $\mathtt{A}$ for a function parameter declared as $\mathit{ref}\ t$. From this, it follows that the corresponding argument in the function call expression must also be a location. In contrast to this, a value of type t must be the argument for a function parameter declared as t (of course

the argument is a location if t is a location type). At the same time, a programming language usually permits a location to be also an argument in this case. Therefore, the rule of construction of the function call expression is as follows: if (f, r, t), where $r = u_1, ..., u_n$ and u_i is either t_i or $ref\ t_i$, is a function declaration and

$$e_i \text{ is } \begin{cases} \text{expression of type } loc(t_i), \\ \qquad\qquad \text{if } u_i \text{ is } ref\ t_i, \\ \text{expression either of type } t_i \text{ or } loc(t_i), \\ \qquad\qquad \text{otherwise,} \end{cases}$$

then $f(e_1, ..., e_n)$ is an expression of type t called the *function call.*

Thus, only a location type expression may be an argument of the function call for a parameter declared as $ref\ t$, and both an expression of type t and an expression of type $loc(t)$ may be an argument in the opposite case. However, the argument of the function itself in the second case may be only a value of type t, i.e., the location has to be dereferenced.

Denote the interpretation of an expression e in algebra $\mathtt{A}$ by $[\![e]\!]^{\mathtt{A}}$. Recall also that an argument expression may be a function call, which, as it was shown in the previous section, yields a pair, an algebra and its element. Therefore, the interpretation of an expression also generally produces such a pair (the same algebra $\mathtt{A}$, if no expression component updates the state). Assume that the function `fst` applied to the pair produces its first component and the function `snd` produces its second component. The interpretation of the function call is then as follows: let $\mathtt{A}_0$ be a Σ-algebra, $\mathtt{A}_1 = \mathtt{fst}([\![e_1]\!]^{\mathtt{A}_0})$, ..., $\mathtt{A}_n = \mathtt{fst}([\![e_n]\!]^{\mathtt{A}_{n-1}})$,

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

then

$$[\![f(e_1, \ldots, e_n)]\!]^{\mathtt{A}_0} = \mathtt{f}_{\mathtt{r}}^{\mathtt{A}_n}(\mathtt{v}_1, \ldots, \mathtt{v}_n).$$

Thus, the interpretation of the function call in algebra $\mathtt{A}$ is the invocation of the corresponding function in algebra $\mathtt{A}_n$ produced by the interpretation of the last argument. The argument is dereferenced when a location instead of a value is passed as argument and used directly in all the other cases. This interpretation rule directly tells the compiler designer how function calls should be compiled.

Example. Let the function f be invoked in block Bl as follows:

```
f(x, m[1])
```

and let $\mathtt{A}$ be a program algebra of the signature Σ and $\mathtt{A}_{\mathtt{Bl}}$ an algebra of the block Bl extending $\mathtt{A}$ with the location constant $\mathtt{x}^{\mathtt{A}_{\mathtt{Bl}}}$. Note that since the function is called in the block Bl containing the declaration of a local variable $\mathtt{x}$, the location associated with $\mathtt{x}$ in algebra $\mathtt{A}_{\mathtt{Bl}}$ will be used in the call; the location associated with $\mathtt{m}$ in algebra $\mathtt{A}_{\mathtt{Bl}}$ remains the same as in algebra $\mathtt{A}$, i.e., $\mathtt{m}^{\mathtt{A}_{\mathtt{Bl}}} = \mathtt{m}^{\mathtt{A}}$. Note also that the algebra produced by evaluation of the actual parameters remains the same algebra $\mathtt{A}_{\mathtt{Bl}}$ since evaluation of the parameters does not update the state. The interpretation of this function call then causes the following function invocation:
$\mathtt{f}^{\mathtt{A}}(\mathtt{x}^{\mathtt{A}_{\mathtt{Bl}}} \uparrow, \mathtt{m}^{\mathtt{A}}(1))$.

6. Updating statements

Let us see how state updates can be used as semantics of the assignment statement and dynamic variable creation statement. The semantics of a statement St interpreted in algebra $\mathtt{A}$ is denoted by $[\![St]\!]^{\mathtt{A}}$. Like the semantics of expressions it generally produces a pair, $\langle \mathtt{A}', \mathtt{v} \rangle$, where $\mathtt{A}'$ is a new algebra and $\mathtt{v}$ is a value. The value may be the label indicated in the goto statement or the value produced by the return statement or the value $\perp$ if the statement does not produce a value.

The assignment statement normally has the following form: $e_l := e_r$, where e_l is an expression of type $loc(t)$ and e_r expression either of type t or of type $loc(t)$.

We first define the semantics of the statement for the cased where t is not an array type assuming that the interpretation of any part of the statement may update the state.

Let $[\![e_l]\!]^{\mathtt{A}} = \langle \mathtt{A}_1, \mathtt{l} \rangle$, $[\![e_r]\!]^{\mathtt{A}_1} = \langle \mathtt{A}_\mathtt{r}, \mathtt{v}_\mathtt{r} \rangle$,

$$\alpha = \begin{cases} (\mathtt{l}, \mathtt{v}_\mathtt{r}), & \text{if } e_r \text{ has type } t, \\ (\mathtt{l}, \mathtt{v}_\mathtt{r} \uparrow), & \text{if } e_r \text{ has type } loc(t), \end{cases}$$

$\mathtt{A}_\mathtt{a} = \mathtt{A}_\mathtt{r}\alpha$. Then $[\![e_l := e_r]\!]^{\mathtt{A}} = \langle \mathtt{A}_\mathtt{a}, \perp \rangle$.

According to this semantics, the following actions should be undertaken when executing the statement:

1) evaluation of the left side of the statement in the source state with its possible update and getting the address of the location to be updated;

2) evaluation of the right side of the statement in the new state with possible its update and getting the updating value or its location;

3) update of the location with getting a new state.

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

$$\Gamma_{\mathtt{a}} = \{(\mathtt{l}_1, \mathtt{v}_1), ..., (\mathtt{l}_n, \mathtt{v}_n)\},$$

where $\mathtt{l}_i = \mathtt{ar1} \uparrow (\mathtt{i})$ and $\mathtt{v}_i = \mathtt{ar2} \uparrow (\mathtt{i}) \uparrow$. Then $[\![ar1 := ar2]\!]^{\mathtt{A}} = \langle \mathtt{A}_{\mathtt{a}}, \bot \rangle$, where $\mathtt{A}_{\mathtt{a}} = \mathtt{A}_{\mathtt{r}}\Gamma$.

In this case, all updates in Γ are fired simultaneously, and each element of the first array is replaced with the corresponding element of the second array. "Simultaneously" means here that the order of the array element update is unimportant, which gives the compiler designer some freedom in choosing the updating algorithm.

The *dynamic variable creation* statement is written in Pascal as $new(p)$, where p is a variable of type (i.e., a pointer). Its formal semantics is as follows.

Let $\delta = (\mathtt{A}_{\mathtt{loc(t)}}, \mathtt{l})$, where $\mathtt{l} \in \mathtt{B}_{\mathtt{Locs}}$ and $\mathtt{l} \notin \mathtt{A}_{\mathtt{loc}}$ (sort update), $\mathtt{A}' = \mathtt{A}\delta$, $\alpha = (\mathtt{p}^{\mathtt{A}}, \mathtt{l})$, and $\mathtt{A}_\alpha = \mathtt{A}'\alpha$. Then $[\![new(p)]\!]^{\mathtt{A}} = \langle \mathtt{A}_\alpha, \bot \rangle$.

Thus, the following actions should be undertaken when executing this statement:

1) allocation of a new location for the sort $\mathtt{A}_{\mathtt{loc(t)}}$;

2) update of the location p by the address of the new location.

If t is an array type $array(ind, t')$, then formally the location sort $loc(t')$ is extended with an extra set $\mathtt{L}$ of locations, the injective finite mapping $\mathtt{B}_{\mathtt{ind}} \rightarrow \mathtt{L}$ is created, and $\mathtt{l}$ is initialized with this mapping. In practice, the address of an array and the address of its first element are normally the same address, and the injective finite mapping $\mathtt{B}_{\mathtt{ind}} \rightarrow \mathtt{L}$ is used just for compiling the indexed variable.

Note that the formal semantics of the statement does not indicate where the new location should be placed in the object program. However, some simple reasoning can help to answer this question. Indeed, it may happen that the variable p is declared in an outer block and the dynamic variable creation statement is executed in an inner block. According to the rules of algebra transformation discussed in Section 3.2, a location dynamically created in a block should exist when the block is exited. It is clear that such a location may not be placed in the stack, otherwise it would be deleted when the block is exited. Thus, we automatically come to the conclusion that a new memory plot, normally called a "heap", should exist in the object program. Note that we did not need to use explicitly this notion in the semantics of the language.

7. Function definition

In the previous sections, we assumed *some function* isof its parameters and result match the types of parameters and result in the function declaration. A real program requires a *definite function* to be associated with a function declaration. In the theory of algebraic specifications [1], they distinguish the notions of *function declaration* (or *function signature*) and *axiom* limiting the set of functions that may be associated with a given function declaration. In programming languages, the function declaration and the axiom are normally combined in a so called *function definition*, which can be formally defined as follows.

Let Σ be a program signature and (f, r, t) a function declaration, where $r = u_1, ..., u_n$ and u_i is either t_i or $ref\, t_i$, $\Delta = \{(p_1, t_1), ..., (p_n, t_n)\}$ is a signature increment, and St a Σ'-block, where Σ' is a block signature constructed with the use of Σ and Δ as described in Section 2.3. Then

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

is a *function definition*. In terms of algebraic specifications, the function definition is just a combination of the function declaration $f(u_1, ..., u_n) : t$ and the axiom $f(p_1, ..., p_n) = St$, where $p_1, ..., p_n$ are algebraic variables that may be used in the statement (function body) St (indeed, the requirement that St is a Σ'-block with the signature increment Δ indicates that these variables may be used in the statement).

Let now $\mathtt{A}$ be a Σ-algebra, $|\mathtt{A}|$ its carrier, and $\xi : \Delta \to |\mathtt{A}|$ a variable valuation such that $\xi(p_i) \in \mathtt{A_t}$ if u_i is t_i, and $\xi(p_i) \in \mathtt{A_{loc(t)}}$ if u_i is $ref\ t_i$. In the same manner as it is described in Section 3.2, basing on algebra $\mathtt{A}$, we construct the Σ'-algebra $\mathtt{A'}$, setting $\mathtt{p_i^{A'}} \uparrow = \xi(\mathtt{p_i})$, if u_i is t_i, and $\mathtt{p_i^{A'}} = \xi(\mathtt{p_i})$ if u_i is $ref\ t_i$. Note that p_i is a local variable initialized with the value of the actual parameter in the first case, and p_i is a local constant equal to the value of the actual parameter (both $\mathtt{p_i^{A'}}$ and $\xi(p_i)$ are the same location) in the second case.

Let now

$$f^{\mathtt{A}}(\xi(p_1), ..., \xi(p_n)) = \langle \mathtt{A_f}, \mathtt{v} \rangle \text{ and } [\![St]\!]^{\mathtt{A'}} = \langle \mathtt{A_{st}}, \mathtt{v_{st}} \rangle.$$

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

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

iff for any Σ-algebra $\mathtt{A}$ and any variable valuation $\xi : \Delta \to |\mathtt{A}|$ it holds:

$$\mathtt{A_f} = \mathtt{A_{st}} \text{ and } \mathtt{v} = \mathtt{v_{st}}.$$

Thus, the semantics of the function definition is a direct instruction for the compiler designer to associate with a function declaration a function

equivalent to the statement used as the function body. Note that *extensional equivalence* (i.e., producing the same results for the same values of the actual parameters) is required. This means that, depending on the choice of the optimization technique, intensionally different functions may be associated with a function declaration; only the result of their execution is important.

Fact. *The semantics of the function definition does not depend on the signature and algebra of the block where the function is called.* Indeed, let $f(e_1, ..., e_n)$ be a function call in signature Σ'' (it may be the signature of some block), where $e_1, ..., e_n$ are Σ''-expressions representing actual parameters of the function call and $\mathtt{A}''$ a Σ''-algebra where the function call is interpreted. We know that the interpretation of a function call in algebra $\mathtt{A}''$ causes the invocation of the corresponding function in algebra $\mathtt{A}''_n$ produced as a result of evaluation of the last argument of the call. Let $\mathtt{v}_1, ..., \mathtt{v}_n$ be the results of evaluation of expressions $e_1, ..., e_n$. Since Σ'' does not includes new data types with respect to Σ, there exists a Σ-algebra $\mathtt{A}$ containing the elements $\mathtt{v}_1, ..., \mathtt{v}_n$ (recall that any set of elements may be associated with a data type) and $\mathtt{f}^{\mathtt{A}''_n}(\mathtt{v}_1, ..., \mathtt{v}_n) = \mathtt{f}^{\mathtt{A}}(\mathtt{v}_1, ..., \mathtt{v}_n)$. The block *St* is interpreted in the algebra extending $\mathtt{A}$ and therefore its interpretation does not depend on $\mathtt{A}''_n$. The fact is proved[1].

Thus, the semantics of the function definition clearly indicates that, when compiling the function body, we ignore the local variables of the block invoking the function and thus avoid the danger of misinterpretation of identifiers redeclared in the block.

Note another instruction for the compiler designer dictated by the semantics of the function definition. The statement *St* is interpreted in the block containing the declarations of the function formal parameters. As we already know, local block variables are allocated in the stack. Therefore, the values of the actual parameters are stored in the stack when the function is invoked, and they are automatically deleted from the stack after the execution of the function body. A value of type *t* is stored in the stack for a function parameter declared as *t*, and an address is stored for a function parameter declared as *ref t*. Passing a location as an argument to the function permits the statement *St* to update the location. Thus, the semantics of the function definition defines the way of compiling the function calls with different kinds of function parameters.

Example. Let the definition of our function *f* be as follows:

```
func f(p1: integer, p2: ref Boolean: char; begin p1 := p1 + 1;
p2 := false; x := 1.5; return 'a' end;
```

According to the semantics of the function definition, the result of the func-

[1] This fact was not recognized by the author when writing the paper [4], and therefore a more complex semantics of the function definition is given in that paper.

tion invocation with some arguments in the algebra A should be the same as the result of the interpretation of the function body in the algebra A extended with the values of these arguments. Let us have an invocation with the arguments v_1 (an integer) and v_2 (a location). The extension of A then produces the algebra A′ with extra elements $p1^{A'}$ and $p2^{A'}$, where $p1^{A'}$ is a location initialized by the value v_1 and $p2^{A'} = v_2$. The signature of this algebra enriches the signature of the algebra A by the elements $(p1, integer)$ and $(p2, Boolean)$ that are used in the process of the function body interpretation. The interpretation will add 1 to the content of the local variable p1, store false in the location $p2^{A'}$ (recall that this is the location of the array element m[1]), and update the content of the global variable x (recall again that the local variable x is forgotten because of the reduct). Following the rules of block interpretation, when the function body is exited, its algebra is reduced to a new algebra of signature Σ and therefore the locations of the formal parameters $p1^{A'}$ and $p2^{A'}$ are deleted from the stack. As a result, the modification of the parameter p1 will not influence the program state while the modification of the parameter p2 will result in modification of the location v_2. In contrast to A, in the new Σ-algebra, A, we have: $v_2 \uparrow = \texttt{false}$ and $x \uparrow = 1.5$. Since the interpretation of the function body produces the pair $\langle A_1, 'a' \rangle$, the function f must also produce the same pair (although an optimizer may simplify the code by deleting the unneeded first statement).

8. Program specification

We can also introduce a notion of constant definition in addition to function definition. So, if (c, t) is a constant declaration and e a Σ_{dat}-expression (constant expression) of type t, then $c : t = e$ is a *constant definition.* A program P(B) satisfies a constant definition $c : t = e$ iff for any Σ-algebra A it holds: $c^A = e^B$.

A *program specification* $\mathcal{PS}$ of program schema $\mathcal{S}$ consists of the definitions of all functions and constants declared in $\mathcal{S}$. A program P(B) satisfies a $\mathcal{PS}$ iff it satisfies the definitions of each function and each constant in $\mathcal{PS}$.

In terms of programming languages, a program specification is just a program text. In Pascal-like languages the program specification is accompanied by a statement, which may be regarded as a term of the program signature that should be evaluated in an algebra satisfying the program specification. In the process of compilation of a program specification, function codes are produced, the symbol table is augmented with function and variable addresses, memory location in the stack and heap are reserved, i.e., all actions needed for the construction of an object program as a model of the specification are performed.

9. Conclusion

It is shown in the paper that the formal semantics of an imperative programming language based on a formal model of the program can serve as an abstract model of some compiler components and algorithms. In particular, it is shown that the program signature is a template of the compiler symbol table, program algebra is a mathematical counterpart of the object program, semantics of function definition and call indicates the mechanisms of their compilation, semantics of blocks requires the stack memory organization, and semantics of dynamic variable creation requires availability of the heap. Thus, the proposed method of formal definition of the semantics of an imperative programming language both provides a way of unambiguous understanding of the language and serves as a good adviser of the compiler designer.

An interesting direction of further research is the use of this kind of semantics for model-based testing of compilers [2]. We hope to proceed with research in this direction.

References

[1] Asteziano E., Kreowski H.-J., Krieg-Brückner B. (Eds.). Algebraic Foundations of System Specification. — Springer, 1999.

[2] El-Far I., Whittaker J. A. Model-based software testing // Encyclopedia of Software Engineering. — 2001. — Vol. 1. — P. 825–837.

[3] Goguen J. A., Malcolm G. Algebraic Semantics of Imperative Programs. — The MIT Press, 1996.

[4] Zamulin A.V. A State-based Semantics of a Pascal-like Language // Proc. Intern. Workshop on Program Understanding. — Novosibirsk, 2003. — P. 47–58.