

Programming by Information-Systems

Peter Bachmann

Brandenburgische Technische Universität Cottbus,

Fakultät 1, Mathematik, Naturwissenschaften und Informatik

Postfach 10 13 44, D-03013 Cottbus

e-mail: pb@informatik.tu-cottbus.de

September 1998

Abstract

The here described approach aims to do programming completely by means of an information system (PIS). The kernel of the PIS is the *database* which contains all information about the program in construction. This database can either be translated into a text of a programming language or interpreted directly.

The interaction with the programmer is done via the *interface* of the PIS. The interface may use routines of the *manager* which implement insertion, deleting, updating and retrieval of data.

In this paper, as a starting point, the structure of the database is defined.

1 Introduction

Currently, *object oriented programming* (oop) is a widely stressed term. The basic idea of it is probably the concept of *classes* connected with *inheritance*. Each class describes the properties of its *instances*, the *objects*. An object consists of some members, namely *data* and *methods*. Members may be *private*. Then, they can only be accessed by the methods of the same object. In such a way, information hiding is implemented. If a class, say *B*, is defined on the basis of another one, say *A*, then *B* inherits all the properties of *A*, i.e. class *B* has at least all the members of *A* and their properties. However, inheritance may be modified by redefinition of methods.

In order to guarantee a well structured design of the aimed software system, hierarchies of predefined classes are built. The programmer should be stimulated to reuse the predefined classes or to define new classes on the basis of them. However, because of the wide variety and great depth of the hierarchies, it is often very complicated to get the right information about needed classes and their properties. Therefore, integrated development environments include tools - often called browsers - for the navigation in the class hierarchies. They allow, for instance, to list all the members of a given class and their properties, all classes of a certain category and so on. Without such tools, the efficient use of class hierarchies is almost impossible.

The collection of these tools may be considered as a kind of programming information system (PIS). The data base of this information system is currently the source text of the program. The tools allow to build fragments of text, to insert and delete fragments from it and so on. Of course, in order to do this efficiently, additional information about the source text must be available. For instance, a lot of additional links to different fragments must be added. Therefore, the source text itself as a linear sequence of signs is not a suitable basis for such an information system.

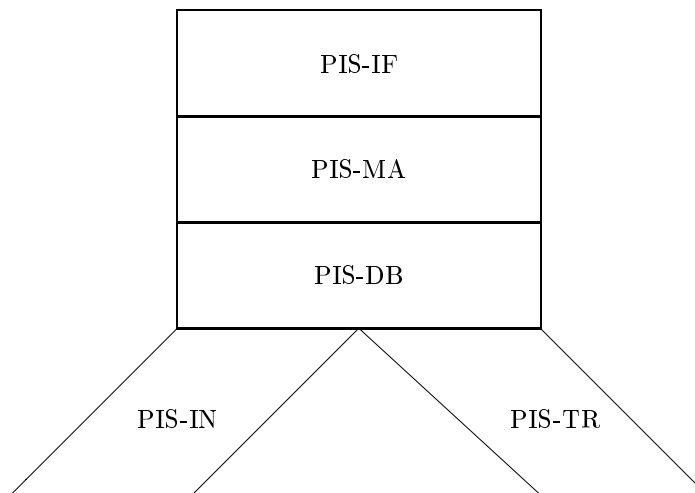
Consequently, our approach is to substitute the source text by a well structured database (the PIS-DB) which completely represents a program system. Since programs are complex objects, the structure of the database will be very interlaced and, may be much more important, complicated integrity constraints have to be fulfilled. So, it is hardly possible to use a relational data model or to work with ER-concepts. Instead, we use structured sets as they are well known from mathematics. The advantage is that the corresponding mathematical notions can be used and only few auxiliary notions must be introduced.

In order to construct a program system, the user feeds the PIS with data. This is done via a (graphical) user interface, the PIS-IF, which also leads the user throughout the construction process. However, because program construction is not a straightforward process, apart from insert also delete- and update-operations are needed, like in an usual information system. Retrieval of information allows to inform about the current state of the product in construction. And, there are incomplete states in the sense that some fragments of the aimed program are not fully defined. There may be holes (error values) in the database. But at the end, all the holes must be filled correctly.

For the navigation in and manipulation of the database the PIS-IF uses algorithms which are collected in a part of the PIS which is called the manager (PIS-MA). The PIS-MA must also guarantee that all the integrity constraints are met. This is probably the main problem to overcome.

When the database is completely built then it describes a program system. To use this system two ways can be gone. Firstly, the database can be interpreted directly. This is done, of course, by an interpreter (PIS-IN). Secondly, by means of a translator (PIS-TR) the database can be transformed into a text of any programming language.

All in all, we get the following scheme for our PIS:



In this paper, the PIS-DB scheme is described. Our objective was to get few clear concepts which can be combined without exceptions. However, we do not follow the paradigm of object oriented programming. The reasons are explained in [1].

In our approach, the type *class* does not exist in the usual sense. Any variable plays the role of an *object*. Of course, values may be structured in different ways and, the several components can be selected. *Functions* (here called *rutines*) are allowed to be values and also components of them. So, we follow the concept of some classical languages like ALGOL68 [2] and OBERON [3].

Inheritance is replaced by two different concepts. Firstly, the *modul* concept taken from MODULAI [4] allows to divide the whole program in a structured system of different parts, to reuse predefined concepts, and to hide information. The global structure of a program system consists of a set of modules. Each of it has an interface, namely the imports and exports. The modules are only connected by the interfaces. The inner part of a module is the definition of identifiers. An identifier can refer to a *type* or a *constant*. Constants are the essential parts of a program, whereas types and literals are used to describe constants.

Secondly, by means of the type system a certain kind of *polymorphism* is simulated. Within object oriented programming, this polymorphism is based on inheritance. Let us assume a class *B* inherits a class *A* and a function has a parameter *p* of class *A*. Then all instances of *B* can also be used as arguments of this parameter. But, there is no way to exclude such behaviour. In our concept, a *pretype* ordering is defined. Inheritance is now an implicate concept. A type *B* inherits (in our sense) a type *A* if *B* is greater than *A*. The special *class A* type contains all the values of those types which are greater than or equal to the base type *A*. In order to allow the above mentioned kind of polymorphism, the parameter *p* has to be of such a class type. Otherwise, if *p* is of type *A*, only values of type *A* are allowed to be arguments.

The type system is mainly used in order to define the necessary constraints of the database. By the type of a constant, its use is restricted. So, for instance, components of the value of a constant may be selected if its type does allow this.

It is to underline, that the PIS-DB scheme, in some sense, is very poor and demands some very strict rules with respect to usual programming languages. For instance, the sets of identifiers occurring in different modules must be disjoint. Or, there are only few and for a first glimpse insufficient constructions to build expressions and statements. However, the PIS-IF may and should enrich the database with additional facilities. So, for instance, unique identifier can be built by the PIS-IF qualifying identifiers with module names. Last ones do not exist in the database but may exist on an external level of the PIS-IF.

For the most parts of the PIS-DB a strict formalization is given. Some parts are left to be undefined, e.g. basic types. Their meaning may depend on special versions of the PIS's. In order to make the paper better legible, all the notions are separately explained in an informal way. So, to get a rough overview, the reader can skip the formal parts except the definitions of syntax which describes the context-free structure of the database.

2 Notions

As already mentioned in the introduction, a program system is a certain finite set of a well-defined structure. In order to define this structure, we use a metalanguage similar to BNF.

In the metalanguage:

- typewriter formatted words mean single elements (e.g. `type`),
- italic formatted words mean sets (e.g. *module*),
- $x := set_construction$ means that set x is defined by a set construction.

A set construction is either simply a set or it defines a set by well-known operations on sets. The here used operations are denoted in the following way:

- $\{x\}$ denotes (as usual) the singleton consisting of the only element x .
- $|a|$ denotes the cardinality of set a .
- $a \cup b$ denotes (as usual) the union of sets a and b .
- $a \times b$ denotes (as usual) the cross product, i.e. a set of ordered pairs.
- $\wp(a)$ denotes the set of subsets of a , $\wp_f(a)$ denotes the set of finite subsets of a .
- $f : a \rightarrow b$ means a total function from set a to set b .
- $b^a := \{f | f : a \rightarrow b\}$ means the set of all total functions from a to b .
- $[a \dashrightarrow b]$ denotes the set of nonempty finite partial functions from set a into set b .
- If $f \in [a \dashrightarrow b]$ then $dom f = \{x | x \in a \wedge \exists y \in b : (x, y) \in f\}$, $im f = \{f(x) | x \in dom f\}$.
- If $f \in [a \dashrightarrow b]$, $g \in [b \dashrightarrow c]$ then $fg \in [a \dashrightarrow c]$ where $(fg)(x) := g(f(x))$.
- \mathbb{N} means the set of natural numbers and, for $n \in \mathbb{N}$ we abbreviate $[n] := \{1, \dots, n\}$ where $[0] := \emptyset$.
- $a^{[n]}$ for $n \in \mathbb{N}$ is also considered as the set of sequences over a of length n , i.e. if $\alpha \in a^{[n]}$ and $i \in \{1, \dots, n\}$ then $\alpha(i) \in a$ is the i th element of the sequence. Obviously, if $\alpha \in a^{[n]}$ then $|\alpha| = n$. We have $a^{[0]} = \{\emptyset\}$, i.e. \emptyset is the empty sequence. Now, by $a^* := \bigcup \{a^{[n]} | n \in \mathbb{N}\}$ we denote the set of all finite sequences over a , $a^+ := a^* - a^{[0]}$ means the set of nonempty finite sequences.
- *tail* cuts the first element from a nonempty sequence, i.e. if for $n \in \mathbb{N} : \alpha \in a^{[n+1]}$ then $tail(\alpha) \in a^{[n]}$ and $tail(\alpha)(i) := \alpha(i+1)$.

- The dot (\cdot) appends an element at the beginning of a sequence, i.e. $(\alpha(1).tail(\alpha)) = \alpha$.

Moreover, if $f \in [A \rightarrow B]$ and $f(i) \in [C \rightarrow D]$ for some $i \in dom f$ then we write shortly $f(ij)$ instead of $f(i)(j)$. In general, if $\alpha \in a^*$ for some set a then $f(\alpha)$ may mean $f(\alpha(1))(\alpha(2)) \dots (\alpha(|\alpha|))$.

Function f may be modified on some argument α by value v . This is denoted by $f[\alpha/v]$, exactly

$$f[\alpha/v](\beta) = \begin{cases} v & \text{if } \alpha = \beta \\ f(\beta) & \text{otherwise} \end{cases}.$$

3 Program-systems

3.1 Syntax

<i>program_system</i>	$:= \wp_f(\text{module})$
<i>module</i>	$:= \text{import} \times \text{definition} \times \text{export}$
<i>import</i>	$:= [\text{identifier} \rightarrow \text{identifier}]$
<i>identifier</i>	$:=$ still undefined
<i>definition</i>	$:= [\text{identifier} \rightarrow (\text{constant} \cup \text{type})]$
<i>constant</i>	$:= \text{type} \times \text{literal}$
<i>export</i>	$:= [\text{identifier} \rightarrow (a_constant \cup \text{type} \cup \{\mathbf{type}\})]$
<i>a_constant</i>	$:= \text{type} \times \{\mathbf{constant}\}$

3.2 Auxiliary notions

Lets $\Pi \in \text{program_system}$ and $m = (i, d, e) \in \Pi$.

The extended definition $\delta_m : \text{identifier} \rightarrow (\text{constant} \cup a_constant \cup \text{type} \cup \{\mathbf{type}\})$ of m is defined by:

$$\delta_m(x) := \begin{cases} d(x) & \text{if } x \in dom d \\ i^*(e'(x')) & \text{if } x \in im i \wedge i(x') = x \wedge x' \in dom e' \end{cases}$$

Such an extended definition can also be defined for the whole program_system Π by:

$$\delta_\Pi(x) := \begin{cases} d(x) & \text{if } \exists (i, d, e) \in \Pi : x \in dom d \\ \delta_\Pi(x') & \text{if } \exists (i, d, e) \in \Pi : x \in im i \wedge i(x') = x \end{cases}$$

Renaming is a function $i^* : a_constant \cup \text{type} \cup \{\mathbf{type}\} \rightarrow a_constant \cup \text{type} \cup \{\mathbf{type}\}$ where

- $i^*(\mathbf{type}) = \mathbf{type}$,
- $(\tau, \mathbf{constant}) \in a_constant \Rightarrow i^*((\tau, \mathbf{constant})) = (i^*(\tau), \mathbf{constant})$,
- for types the function i^* is defined in 4.2.

Containing is a function $\Xi : \text{constant} \cup a_constant \cup \text{type} \cup \text{literal} \cup \{\mathbf{type}\} \rightarrow \wp_f(\text{identifier})$ where

- $\Xi(\mathbf{type}) = \Xi(\mathbf{constant}) = \emptyset$
- $(\tau, l) \in \text{constant} \cup a_constant \Rightarrow \Xi((\tau, l)) = \Xi(\tau) \cup \Xi(l)$
- for types and literals the function Ξ is defined in 4.2 and 5.2 respectively.

Compatibility is a relation $\triangleright \subseteq \text{literal} \times \text{type}$ defined in 5.2.

Similarity \sim_Π is defined in 4.2.

3.3 Constraints

Let Π any program-system and $m = (i, d, e) \in \Pi$. Then it must hold:

- i is injective
- $\text{dom } i \cap \text{im } i = \emptyset$
- the transitive closure of relation $\bigcup\{i \mid (i, d, e) \in \Pi\}$ is irreflexive
- $\text{dom } i \subseteq \bigcup\{\text{dom } e' \mid (i', d', e') \in \Pi\}$
- $x \in \text{dom } i \cap \text{dom } e' \Rightarrow \Xi(e'(x)) \subseteq \text{dom } i$
- $\text{im } i \cap \text{dom } d = \emptyset$
- $x \in \text{dom } d \wedge d(x) = (\tau, l) \in \text{constant} \Rightarrow l \triangleright \tau$
- $\text{dom } e \subseteq \text{im } i \cup \text{dom } d$
- $x \in \text{dom } e \Rightarrow \Xi(e(x)) \subseteq \text{dom } e$
- $x \in \text{dom } e \wedge e(x) = (\tau, \text{constant}) \in \text{a_constant} \Rightarrow \delta_m(x) = (\tau, l) \in \text{constant} \cup \text{a_constant}$
- $x \in \text{dom } e \wedge e(x) \in \text{type} \Rightarrow \delta_m(x) = e(x)$
- $x \in \text{dom } e \wedge e(x) = \text{type} \Rightarrow \delta_m(x) \in \text{type} \cup \{\text{type}\}$
- $\forall m' = (i', d', e') \in \Pi : (\text{im } i \cup \text{dom } d) \cap (\text{im } i' \cup \text{dom } d') \neq \emptyset \Rightarrow m = m'$

3.4 Semantics

Semantics is a function $\llbracket \cdot \rrbracket : S \rightarrow M$ which assigns meanings to the elements of sets defined by syntax. Meanings may differ from set to set. The basis of all meanings is a set U (the *Universe* of values) which fulfills the following conditions:

- $\emptyset \in U$,
- $\mathbb{N} \subseteq U$,
- $U^* \subseteq U$,
- $[\text{index} \rightarrow U] \subseteq U$,
- $\text{index} \cap U = \emptyset$,
- $\forall \tau \in \text{type} - \{\text{void}\} \exists V_\tau \subseteq U : |V_\tau| = \aleph_0$,
- $V_{\text{void}} = \{\underline{\text{nil}}\}$,
- $\tau \sim_\Pi \tau' \Rightarrow V_\tau = V_{\tau'}$,
- $V_\tau \cap V_{\tau'} \neq \emptyset \Rightarrow \tau \sim_\Pi \tau'$,

Semantics of *program_systems* is defined as:

- $\Pi \in \text{program_system} \Rightarrow \llbracket \Pi \rrbracket = \bigcup\{\llbracket m \rrbracket \mid m \in \Pi\}$
- $m = (i, d, e) \in \text{module} \Rightarrow \llbracket m \rrbracket = \{(x, \llbracket d(x) \rrbracket) \mid x \in \text{dom } d \wedge d(x) \in \text{constant}\}$
- $c = (\tau, l) \in \text{constant} \Rightarrow \llbracket c \rrbracket = \llbracket l \rrbracket$

3.5 Explanations

A *program_system* consists of a set of modules. Each *module* $m = (i, d, e)$ has three parts: an *import* i , a *definition* d , and an *export* e .

A definition assigns *constants* and *types* to *identifiers*, i.e. if x is an identifier then $d(x)$ is a constant or a type.

By the import, definitions from other modules are available in the own module. But, by renaming an imported identifier becomes always an identifier of the module from which it is imported. So, if i is the import of module m and $x \in \text{dom } i$ then x is an identifier which is exported by some other modul and $i(x)$ is an identifier of module m . Only the renamed identifier $i(x)$ can be used within module m .

The definition of module m is extended by the import, where we get the *extended definition* δ_m . Any identifier occuring within a definition which is imported must be imported too! By the import, a renaming of all the imported identifiers is done. Of course, an identifier cannot be defined via the import and by the definition at the same time.

The export makes parts of the extended definition available outside of the module. That means, also an import can be exported. However, cycles in this process are forbidden. Therefore, an identifier must be defined directly or indirectly via some renamings within a definition of a module.

For a *constant*, only its type is exported. Then, the constant becomes an *a_constant*. For a *type*, the definition of an identifier can only be changed by hiding. In this case, $e(x) \in \{\mathbf{type}\}$ which means that outside of the module only the role of the identifier as a type is known but nothing about the details. In this case, every constant of this type can only be used in an restricted manner.

Every identifier which is defined anywhere belongs to exactly one module. Different modules possess disjoint sets of identifiers.

Constants are the essential parts of a *program_system*. A constant is described by a type and a literal. In order to make this description consistent, the literal must be compatible to the type. This is explained in detail in 5.2.

The meaning of a literal is a value, explained in 5.5. The meaning of a constant is the meaning of the literal which describes its value. The meaning of a module is a function which assigns to each identifier defined as a constant a value namely the meaning of the literal of this constant. And, at the end, the meaning of a *program_system* is the union of all those functions which are meanings of modules.

All values are collected in the set U , the *universe*. The universe must contain the empty set as an element, the set of natural numbers (including zero) as a subset and must be closed with respect to sequences of values and functions from *indizes* to values. Note, that *indizes* are special elements which do not be values!

Finally, for every type τ there is a special set V_τ of values, called *variables* of type τ . Each of it is infinitely countable, except the set V_{void} which consists only of the special variable **nil**. If two types are *similar* then the corresponding sets of variables coincide and if two sets of variables are not disjoint then the types of them have to be similar.

In a computation process, there exists a function φ which assigns values to some variables. If v is a variable of type τ , then $v \in V_\tau$ and $\varphi(v)$ must be of type τ .

4 Types

4.1 Syntax

$type$	$:=$	$defined_type \cup basic_type \cup void_type \cup$ $tuple_type \cup array_type \cup list_type \cup struct_type \cup$ $union_type \cup class_type \cup$ $variable_type \cup routine_type$
$defined_type$	$:=$	$identifier$
$basic_type$	$:=$	still undefined
$void_type$	$:=$	$\{\mathbf{void}\}$
$tuple_type$	$:=$	$\{\mathbf{tuple}\} \times type^+$
$array_type$	$:=$	$\{\mathbf{array}\} \times number \times type$
$number$	$:=$	$\mathbb{N} - \{0\}$
$list_type$	$:=$	$\{\mathbf{list}\} \times type$
$struct_type$	$:=$	$\{\mathbf{struct}\} \times [index \rightarrow type]$
$index$	$:=$	still undefined
$union_type$	$:=$	$\{\mathbf{union}\} \times \wp_f(type)$
$class_type$	$:=$	$\{\mathbf{class}\} \times type$
$variable_type$	$:=$	$\{\mathbf{var}\} \times type$
$routine_type$	$:=$	$\{\mathbf{rout}\} \times type \times type$

4.2 Auxiliary notions

Let Π be any program_system, $m = (i, d, e) \in \pi$.

Renaming i^* is defined for types in the following way:

- $x \in defined_type \Rightarrow i^*(x) = i(x)$
- $\tau \in basic_type \Rightarrow i^*(\tau) = \tau$
- $i^*(\mathbf{void}) = \mathbf{void}$
- $\tau \in type^+ \wedge \tau' \in type^+ \wedge dom \tau = dom \tau' \wedge \forall x \in dom \tau' : \tau'(x) = i^*(\tau(x))$
 $\Rightarrow i^*((\mathbf{tuple}, \tau)) = (\mathbf{tuple}, \tau')$
- $n \in number \wedge \tau \in type \Rightarrow i^*((\mathbf{array}, n, \tau)) = (\mathbf{array}, n, i^*(\tau))$
- $\tau \in [index \rightarrow type] \wedge \tau' \in [index \rightarrow type] \wedge dom \tau = dom \tau' \wedge \forall x \in dom \tau' : \tau'(x) = i^*(\tau(x))$
 $\Rightarrow i^*((\mathbf{struct}, \tau)) = (\mathbf{struct}, \tau')$
- $\tau \in \wp_f(type) \Rightarrow i^*((\mathbf{union}, \tau)) = (\mathbf{union}, \{i^*(\tau') \mid \tau' \in \tau\})$
- $\tau \in type \Rightarrow i^*((\mathbf{class}, \tau)) = (\mathbf{class}, i^*(\tau))$
- $\tau \in type \Rightarrow i^*((\mathbf{var}, \tau)) = (\mathbf{var}, i^*(\tau))$
- $\tau, \tau' \in type \Rightarrow i^*((\mathbf{rout}, \tau, \tau')) = (\mathbf{rout}, i^*(\tau), i^*(\tau'))$

Containing Ξ is defined for types in the following way:

- $x \in defined_type \Rightarrow \Xi(x) = \{x\}$
- $\tau \in basic_type \Rightarrow \Xi(\tau) = \emptyset$
- $\Xi(\mathbf{void}) = \emptyset$
- $\tau \in type^+ \Rightarrow \Xi((\mathbf{tuple}, \tau)) = \bigcup \{\Xi(\tau(x)) \mid x \in dom \tau\}$
- $n \in number \wedge \tau \in type \Rightarrow \Xi((\mathbf{array}, n, \tau)) = \Xi(\tau)$

- $\tau \in [index \rightarrow type] \Rightarrow \Xi((\mathbf{struct}, \tau)) = \bigcup \{\Xi(\tau(x)) \mid x \in dom \tau\}$
- $\tau \in \wp_f(type) \Rightarrow \Xi((\mathbf{union}, \tau)) = \bigcup \{\Xi(\tau') \mid \tau' \in \tau\}$
- $\tau \in type \Rightarrow \Xi((\mathbf{class}, \tau)) = \Xi(\tau)$
- $\tau \in type \Rightarrow \Xi((\mathbf{var}, \tau)) = \Xi(\tau)$
- $\tau, \tau' \in type \Rightarrow \Xi((\mathbf{rout}, \tau, \tau')) = \Xi(\tau) \cup \Xi(\tau')$

Embedding is the least transitive relation $\Subset \subseteq type \times type$ defined by:

- $x \in \Xi(\delta_\Pi(y)) \Rightarrow x \Subset y$

The subtype-relation $\lesssim_m \subseteq type \times type$ is defined for each module $m \in \Pi$ to be the least transitive relation which meets the following assertions:

- $\tau \in type \Rightarrow \tau \lesssim_m \tau$
- $x \in dom \delta_m \wedge \delta_m(x) \in type \Rightarrow x \sim_m \delta_m(x)$
- $\tau \in type \Rightarrow (\mathbf{var}, \mathbf{void}) \lesssim_m (\mathbf{var}, \tau)$
- $\tau, \tau' \in type^+ \wedge |\tau| = |\tau'| \wedge \forall n \in \{1, \dots, |\tau|\} : \tau(n) \lesssim_m \tau'(n) \Rightarrow (\mathbf{tuple}, \tau) \lesssim_m (\mathbf{tuple}, \tau')$
- $\tau, \tau' \in type \wedge \tau \lesssim_m \tau' \wedge n \in number \Rightarrow (\mathbf{array}, n, \tau) \lesssim_m (\mathbf{array}, n, \tau')$
- $\tau, \tau' \in type \wedge \tau \lesssim_m \tau' \Rightarrow (\mathbf{list}, \tau) \lesssim_m (\mathbf{list}, \tau')$
- $\tau, \tau' \in [index \rightarrow type] \wedge dom \tau = dom \tau' \wedge \forall x \in dom \tau : \tau(x) \lesssim_m \tau'(x) \Rightarrow (\mathbf{struct}, \tau) \lesssim_m (\mathbf{struct}, \tau')$
- $\tau' \in \wp_f(type) \wedge \tau \in \tau' \Rightarrow \tau \lesssim_m (\mathbf{union}, \tau')$
- $\tau, \tau' \in \wp_f(type) \wedge \forall t \in \tau \exists t' \in \tau' : t \lesssim_m t' \Rightarrow (\mathbf{union}, \tau) \lesssim_m (\mathbf{union}, \tau')$
- $\tau, \tau' \in type \wedge \tau \sqsubseteq_m \tau' \Rightarrow \tau' \lesssim_m (\mathbf{class}, \tau)$
- $\tau, \tau' \in type \wedge \tau \sqsubseteq_m \tau' \Rightarrow (\mathbf{class}, \tau') \lesssim_m (\mathbf{class}, \tau)$
- $\tau, \tau' \in type \wedge \tau \lesssim_m \tau' \Rightarrow (\mathbf{class}, \tau) \lesssim_m (\mathbf{class}, \tau')$
- $\tau_1, \tau_2, \tau'_1, \tau'_2 \in type \wedge \tau'_1 \lesssim_m \tau_1 \wedge \tau_2 \lesssim_m \tau'_2 \Rightarrow (\mathbf{rout}, \tau_1, \tau_2) \lesssim_m (\mathbf{rout}, \tau'_1, \tau'_2)$

The similar-relation \sim_m is the equivalence induced by \lesssim_m , i.e.

$$\tau \sim_m \tau' \text{ if and only if } \tau \lesssim_m \tau' \text{ and } \tau' \lesssim_m \tau.$$

The similar-relation is extended to a whole program_system Π by defining \sim_Π as the transitive closure of $\bigcup \{\sim_m \mid m \in \Pi\}$.

The pretype-relation $\sqsubseteq_m \subseteq type \times type$ is defined for each module $m \in \Pi$ to be the least transitive relation which meets the following assertions:

- $\tau \in type \Rightarrow \tau \sqsubseteq_m \tau$
- $\tau, \tau' \in type^+ \wedge dom \tau \subseteq dom \tau' \wedge \forall n \in dom \tau : \tau(n) \sim_m \tau'(n) \Rightarrow (\mathbf{tuple}, \tau) \sqsubseteq_m (\mathbf{tuple}, \tau')$
- $m, n \in number \wedge m \leq n \Rightarrow (\mathbf{array}, m, \tau) \sqsubseteq_m (\mathbf{array}, n, \tau)$
- $\tau, \tau' \in [index \rightarrow type] \wedge dom \tau \subseteq dom \tau' \wedge \forall x \in dom \tau : \tau(x) \sim_m \tau'(x) \Rightarrow (\mathbf{struct}, \tau) \sqsubseteq_m (\mathbf{struct}, \tau')$
- $\tau \sqsubseteq_m \tau' \wedge \tau' \sim_m \tau'' \Rightarrow \tau \sqsubseteq_m \tau''$

4.3 Constraints

Let Π be any program_system. Then,

- \Subset must be irreflexive

and, for each module $m \in \Pi$ it must hold:

- $x \in \text{dom } \delta_m \wedge (\delta_m(x) = (\tau, l) \in \text{constant} \cup \text{a_constant} \vee \delta_m(x) = \tau \in \text{type}) \wedge y \in \Xi(\tau)$
 $\Rightarrow y \in \text{dom } \delta_m \wedge \delta_m(y) \in (\text{type} \cup \{\text{type}\})$

4.4 Semantics

For any program_system Π the function $\llbracket \cdot \rrbracket : \text{type} \rightarrow \wp(U)$ assigns a certain subset $\llbracket \tau \rrbracket$ of universe U to a type τ . Function $\llbracket \cdot \rrbracket$ has to fulfill the following conditions:

- $x \in \text{defined_type} \Rightarrow \llbracket x \rrbracket = \llbracket \delta_\Pi(x) \rrbracket$
- $\tau \in \text{basic_type} \Rightarrow \llbracket \tau \rrbracket \subseteq U_B$
 where $U_B = U - \bigcup \{V_\tau \mid \tau \in \text{type}\} - \{\emptyset\} - U^* - [\text{index} \twoheadrightarrow U]$,
- $\llbracket (\text{var}, \text{void}) \rrbracket = \{\underline{\text{nil}}\}$,
- $\tau, \tau' \in \text{basic_type} \wedge \llbracket \tau \rrbracket \cap \llbracket \tau' \rrbracket \neq \emptyset \Rightarrow \tau = \tau'$,
- $\llbracket \text{void} \rrbracket := \{\emptyset\}$,
- $\tau \in \text{type}^{[n]} \Rightarrow \llbracket (\text{tuple}, \tau) \rrbracket = \{\alpha \mid \alpha \in U^{[n]} \wedge \forall m \in [n] : \alpha(m) \in \llbracket \tau(m) \rrbracket\}$,
- $n \in \text{number} \wedge \tau \in \text{type} \Rightarrow \llbracket (\text{array}, n, \tau) \rrbracket = \llbracket \tau \rrbracket^{[n]}$,
- $\tau \in \text{type} \Rightarrow \llbracket (\text{list}, \tau) \rrbracket := \llbracket \tau \rrbracket^*$,
- $\tau \in [\text{index} \twoheadrightarrow \text{type}]$
 $\Rightarrow \llbracket (\text{struct}, \tau) \rrbracket = \{\alpha \mid \alpha \in [\text{index} \twoheadrightarrow U] \wedge \text{dom } \alpha = \text{dom } \tau \wedge \forall x \in \text{dom } \alpha : \alpha(x) \in \llbracket \tau(x) \rrbracket\}$,
- $\tau \in \wp_f(\text{type}) \Rightarrow \llbracket (\text{union}, \tau) \rrbracket := \bigcup \{\llbracket \tau' \rrbracket \mid \tau' \in \tau\}$,
- $\tau \in \text{type} \Rightarrow \llbracket (\text{class}, \tau) \rrbracket := \bigcup \{\llbracket \tau' \rrbracket \mid \tau \sqsubseteq \tau'\}$,
- $\tau \in \text{type} \Rightarrow \llbracket (\text{var}, \tau) \rrbracket := V_\tau \cup \{\underline{\text{nil}}\}$,
- $\tau, \tau' \in \text{type} \Rightarrow \llbracket (\text{rout}, \tau, \tau') \rrbracket := [\Phi \times T \twoheadrightarrow \Phi \times T']$
 where $\llbracket \tau \rrbracket \subseteq T$, $T' \subseteq \llbracket \tau' \rrbracket$, and $\Phi := \bigcup \{V_\tau \twoheadrightarrow \llbracket \tau \rrbracket \mid \tau \in \text{type}\} \cup \{(\underline{\text{nil}}, \emptyset)\}$.

4.5 Explanations

The meaning of a type is a set of values. Within a definition, a type is assigned to an identifier which can be used instead of the type. This identifier is called a *defined_type*.

The following types are available:

- *basic_types* which are not defined in detail. The intention of *basic_types* is that the values of them are the information units of the computer like fixed point and floating point numbers of different length. *Basic_types* are units not composed from other ones.
- *void_type* has as meaning the singleton consisting of the empty set. It represents *nothing*.
- A *tuple_type* is based on a nonempty sequence of types, say τ . The meaning of it is a set of tuples, all of the same kind. The number of elements of each tuple equals to the length of τ . The type of the m -th component of each tuple is $\tau(m)$, the m -th element of sequence τ . Here, like in mathematics, a tuple is considered to be a function α which assigns to each position m a value $\alpha(m)$. This value must be an element of the meaning of the type $\tau(m)$, i.e. $\alpha(m) \in \llbracket \tau(m) \rrbracket$.
- An *array_type* is based on a number, say n and on a type, say τ . The meaning of it is also a set of tuples with n components all of type τ .

- A *list_type* is based on a type, say τ . The meaning of it is also a set of tuples with any number of components but all of the same type τ .
- A *struct_type* is based on a function, say τ , from a certain finite set of indices to a set of types. The meaning of it is a generalized tuple where the components are not selected by numbers but by indices. If α is such a value then for an index $x \in \text{dom } \alpha$ the component $\alpha(x)$ must be a value of type $\tau(x)$. This corresponds to the well-known **record**-types of languages like PASCAL where the names of the components are here called indices.
- A *union_type* is based on a finite set of types, say τ . The meaning of it is the union of the meanings of all the members of τ .
- A *class_type* is based on a type, say τ . The meaning of it is the union of the meanings of all types τ' for which τ is a pretype ($\tau \sqsubseteq_m \tau'$). The aim of the pretype-relation \sqsubseteq_m is explained below.
- A *variable_type* is based on a type, say τ . The meaning of it is the set V_τ of variables (see 3.5).
- A *routine_type* is based on two types, say τ, τ' . The meaning of it is a set of routines (subprograms, functions,...). Each routine is a partial function which has two arguments: the first one, say φ , is an element from set Φ , and the second one, say p , is a value from set $\llbracket \tau \rrbracket$. The result, if any, is a pair (φ', p') where φ' is a variable-assignment and p' is a value from $\llbracket \tau' \rrbracket$. p is called the parameter of type τ , p' is called the result-value of type τ' . Types τ as well as τ' may be **void**. In this case, no argument or/and no result-value exists. Elements φ from set Φ are called variable-assignments. They assign a value $\varphi(v) \in \llbracket \hat{\tau} \rrbracket$ to each variable $v \in V_{\hat{\tau}} \cap \text{dom } \varphi$. For all variable-assignments φ , always $\varphi(\underline{\text{nil}}) = \emptyset$ holds.

The formal definition of renaming just means that all the identifiers which are contained in a type construction are changed in accordance to the import renaming i .

The function Ξ (containing) yields all those identifiers included in a type.

Relation embedding (\Subset) expresses whether a certain identifier x is directly or indirectly - also by export and import via some modules - used in a type construction. Since embedding has to be irreflexive, any type-construction cannot be based on itself. However, this does not hold for variable-types. This exception allows to build cycles in data-structures.

The subtype relation \lesssim_m is an important notion to formulate restrictions for the construction of routine literals, especially for expressions (see 5.5). If $\tau \lesssim_m \tau'$ then $\llbracket \tau \rrbracket \subseteq \llbracket \tau' \rrbracket$. That means, if $\tau \lesssim_m \tau'$ and $\tau' \lesssim_m \tau$ then $\llbracket \tau \rrbracket = \llbracket \tau' \rrbracket$. In this case, τ and τ' are called similar ($\tau \sim_m \tau'$).

If an identifier x is defined as type ($x \in \delta_m(x) \wedge \delta_m(x) = \tau \in \text{type}$) then x is similar to τ . It can be used instead of τ . However, if $\delta_m(x) = \text{type}$ then no details about the type of x are known and therefore, x is only similar to itself and to all identifiers y with $\delta_m(y) = x$. This is possible by an export of an identifier x' with hidden details ($e'(x') = \text{type}$) where x' is imported by module $m = (i, d, e)$ and renamed to x ($i(x') = x$). In this way, the subtype-relation and the similarity may depend on the module m .

By extending the similarity on the whole program-system (\sim_Π) all details of any type-identifiers are given. This is necessary to define semantics.

The pretype relation \sqsubseteq_m is used to defined the meaning of a *class_type*. If $\tau \sqsubseteq_m \tau'$ and $\tau \neq \tau'$ then the meaning of τ as well as of τ' are functions defined by *tuple_types*, *array_types* or *struct_types*. And, if $\alpha \in \llbracket \tau \rrbracket$, $\alpha' \in \llbracket \tau' \rrbracket$ then $\text{dom } \alpha \subseteq \text{dom } \alpha'$ and the types of all components of α coincide with the types of the corresponding components of α' . Therefore, all operations which can be applied to any component of α can be applied to the corresponding components of α' too. If the parameter of a routine has type (*class*, τ) then also values of type τ' can be used as argument.

5 Literals

5.1 Syntax

<i>literal</i>	$:=$	$defined_constant \cup basic_literal \cup variable_literal \cup literal^+ \cup [index \rightarrow literal] \cup routine_literal$
<i>defined_constant</i>	$:=$	$\{null, nil\} \cup identifier$
<i>basic_literal</i>	$:=$	still undefined
<i>variable_literal</i>	$:=$	$\{anonymous\}$
<i>routine_literal</i>	$:=$	$local_variable \times body$
<i>local_variable</i>	$:=$	$variable_type$
<i>body</i>	$:=$	$statement^*$
<i>statement</i>	$:=$	$exp_statement \cup switch_statement \cup loop_statement \cup break_statement \cup return_statement$
<i>exp_statement</i>	$:=$	$\{expr\} \times expression$
<i>expression</i>	$:=$	$\{loc, par\} \cup defined_constant \cup generation \cup dereferencing \cup assignment \cup construction \cup selection \cup application$
<i>generation</i>	$:=$	$\{new\} \times type$
<i>dereferencing</i>	$:=$	$\{deref\} \times expression$
<i>assignment</i>	$:=$	$\{assign\} \times expression \times target$
<i>target</i>	$:=$	$expression \times qualification$
<i>qualification</i>	$:=$	$(index \cup number \cup expression)^*$
<i>construction</i>	$:=$	$\{tuple, array, list\} \times (expression^+) \cup \{struct\} \times (index \times expression)^+$
<i>selection</i>	$:=$	$\{select\} \times expression \times (index \cup number \cup expression)$
<i>application</i>	$:=$	$\{apply\} \times expression \times expression$
<i>switch_statement</i>	$:=$	$\{switch\} \times expression \times (statement^*)^+$
<i>loop_statement</i>	$:=$	$\{loop\} \times statement^*$
<i>break_statement</i>	$:=$	$\{break\} \times number$
<i>return_statement</i>	$:=$	$\{return\} \times expression$

5.2 Auxiliary notions

Let Π be any program_system, $m = (i, d, e) \in \pi$.

Containing Ξ is defined for literals in the following way:

- $x \in defined_constant \Rightarrow \Xi(x) = \{x\}$
- $l \in basic_literal \Rightarrow \Xi(l) = \emptyset$
- $\Xi(anonymous) = \emptyset$
- $\lambda \in literal^+ \Rightarrow \Xi(\lambda) = \bigcup \{\Xi(\lambda(n)) \mid n \in dom \lambda\}$
- $\lambda \in [index \rightarrow literal] \Rightarrow \Xi(\lambda) = \bigcup \{\Xi(\lambda(x)) \mid x \in dom \lambda\}$
- $(v, b) \in routine_literal \Rightarrow \Xi((v, b)) = \emptyset$

Embedding is the least transitive relation $\Subset \subseteq literal \times literal$ defined by:

- $\delta_{\Pi}(y) = (\tau, l) \in constant \wedge x \in \Xi(l) \Rightarrow x \Subset y$

Compatibility is a relation $\triangleright \subseteq \text{literal} \times \text{type}$ defined by:

- $x \in \text{defined_constant} \wedge \delta_{\Pi}(x) = (\tau, l) \in \text{constant} \Rightarrow x \triangleright \tau$
- $\text{null} \triangleright \text{void}$
- $\tau \in \text{type} \Rightarrow \text{nil} \triangleright (\text{var}, \tau)$
- $\text{anonymous} \triangleright (\text{var}, \tau)$
- $l \triangleright \tau \wedge \tau \sim_{\Pi} \tau' \Rightarrow l \triangleright \tau'$
- $l \in \text{literal}^+ \wedge \tau \in \text{type}^+ \wedge \text{dom } l = \text{dom } \tau \wedge \forall n \in \text{dom } l : l(n) \triangleright \tau(n) \Rightarrow l \triangleright (\text{tuple}, \tau)$
- $l \in \text{literal}^+ \wedge \forall n \in \text{dom } l : l(n) \triangleright \tau \Rightarrow l \triangleright (\text{array}, |l|, \tau)$
- $l \in \text{literal}^+ \wedge \forall n \in \text{dom } l : l(n) \triangleright \tau \Rightarrow l \triangleright (\text{list}, \tau)$
- $l \in [\text{index} \dashrightarrow \text{literal}] \wedge \tau \in [\text{index} \dashrightarrow \text{type}] \wedge \text{dom } l = \text{dom } \tau \wedge \forall n \in \text{dom } l : l(n) \triangleright \tau(n) \Rightarrow l \triangleright (\text{struct}, \tau)$
- $\tau \in \wp_f(\text{types}) \wedge \exists t \in \tau : l \triangleright t \Rightarrow l \triangleright (\text{union}, \tau)$
- $\tau \sqsubseteq \tau' \wedge l \triangleright \tau' \Rightarrow l \triangleright (\text{class}, \tau)$
- $\sigma \in \text{body} \wedge (\tau_1, \tau_2, \tau_3) \models \sigma \Rightarrow (\tau_1, \sigma) \triangleright (\text{rout}, \tau_2, \tau_3)$

Type inference $\vdash_{m, \tau_1, \tau_2, \tau_3} \subseteq \text{expression} \times \text{type}$ is defined for each module $m \in \Pi$ and three given types τ_1, τ_2, τ_3 . In order to shorten the notation, the subscripts are omitted. It is defined by:

- $\text{loc} \vdash \tau_1$
- $\text{par} \vdash \tau_2$
- $x \in \text{dom } \delta_m \wedge \delta_m(x) = (\tau, l) \Rightarrow x \vdash \tau$
- $\text{null} \vdash \text{void}$
- $\text{nil} \vdash (\text{var}, \text{void})$
- $\tau \in \text{type} \Rightarrow (\text{new}, \tau) \vdash (\text{var}, \tau)$
- $e \vdash (\text{var}, \tau) \Rightarrow (\text{deref}, e) \vdash \tau$
- $e \vdash \tau \Rightarrow (\text{assign}, e, t) \vdash \tau$
- $e \in \text{expression}^+ \wedge \tau \in \text{type}^+ \wedge \forall n \in \text{dom } e : e(n) \vdash \tau(n) \Rightarrow (\text{tuple}, e) \vdash (\text{tuple}, \tau)$
- $e \in \text{expression}^+ \wedge e(1) \vdash \tau \Rightarrow (\text{array}, e) \vdash (\text{array}, |e|, \tau)$
- $e \in \text{expression}^+ \wedge e(1) \vdash \tau \Rightarrow (\text{list}, e) \vdash (\text{list}, \tau(1))$
- $s \in (\text{index} \times \text{expression})^+ \wedge \tau \in [\text{index} \dashrightarrow \text{type}] \wedge |s| = |\tau| \wedge (\forall x \in \text{dom } \tau \exists n \in \text{dom } s : s(n) = (x, e) \wedge e \vdash \tau(x)) \Rightarrow (\text{struct}, s) \vdash (\text{struct}, \tau)$
- $e \vdash (\text{struct}, \tau) \wedge x \in \text{dom } \tau \Rightarrow (\text{select}, e, x) \vdash \tau(x)$
- $e \vdash (\text{tuple}, \tau) \wedge n \in \text{dom } \tau \Rightarrow (\text{select}, e, n) \vdash \tau(n)$
- $e \vdash (\text{array}, n, \tau) \Rightarrow (\text{select}, e, e') \vdash \tau$
- $e \vdash (\text{list}, \tau) \Rightarrow (\text{select}, e, e') \vdash \tau$

- $e \vdash (\mathbf{rout}, \tau, \tau') \Rightarrow (\mathbf{apply}, e, e') \vdash \tau'$
- $e \vdash \tau \wedge \tau \sim_m \tau' \Rightarrow e \vdash \tau'$

The sign \vdash is also used in order to describe a relation $\vdash_{m, \tau_1, \tau_2, \tau_3} \subseteq (\text{type} \times \text{qualification}) \times \text{type}$, defined by:

- $(\tau, \emptyset) \vdash \tau$
- $\tau \in [\text{index} \rightarrow \text{type}] \wedge x \in \text{dom } \tau \wedge (\tau(x), \alpha) \vdash \tau' \Rightarrow ((\mathbf{struct}, \tau), x.\alpha) \vdash \tau'$
- $\tau \in \text{type}^+ \wedge n \in \text{dom } \tau \wedge (\tau(n), \alpha) \vdash \tau' \Rightarrow ((\mathbf{tuple}, \tau), n.\alpha) \vdash \tau'$
- $\tau \in \text{type} \wedge n \in \text{number} \wedge e \in \text{expression} \wedge (\tau, \alpha) \vdash \tau' \Rightarrow ((\mathbf{array}, n, \tau), e.\alpha) \vdash \tau'$
- $\tau \in \text{type} \wedge e \in \text{expression} \wedge (\tau, \alpha) \vdash \tau' \Rightarrow ((\mathbf{list}, \tau), e.\alpha) \vdash \tau'$
- $(\tau, \alpha) \vdash \tau' \wedge \tau \sim_m \tau'' \Rightarrow (\tau'', \alpha) \vdash \tau'$
- $(\tau, \alpha) \vdash \tau' \wedge \tau' \sim_m \tau'' \Rightarrow (\tau, \alpha) \vdash \tau''$

Soundness $\dashv_{m, \tau_1, \tau_2, \tau_3} \subseteq \text{type} \times \text{qualification}$ is defined for each module $m \in \Pi$ and three given types τ_1, τ_2, τ_3 . In order to shorten the notation, the subscripts are omitted. It is defined by:

- $\tau \dashv \emptyset$
- $\tau \in [\text{index} \rightarrow \text{type}] \wedge x \in \text{dom } \tau \wedge \tau(x) \dashv \alpha \Rightarrow (\mathbf{struct}, \tau) \dashv x.\alpha$
- $\tau \in \text{type}^+ \wedge n \in \text{dom } \tau \wedge \tau(n) \dashv \alpha \Rightarrow (\mathbf{tuple}, \tau) \dashv n.\alpha$
- $\tau \in \text{type} \wedge n \in \text{number} \wedge e \in \text{expression} \wedge \tau \dashv \alpha \Rightarrow (\mathbf{array}, n, \tau) \dashv e.\alpha$
- $\tau \in \text{type} \wedge e \in \text{expression} \wedge \tau \dashv \alpha \Rightarrow (\mathbf{list}, \tau) \dashv e.\alpha$
- $\tau \dashv \alpha \wedge \tau' \sim_m \tau \Rightarrow \tau' \dashv \alpha$

Correctness wrt. a module $m \in \Pi$ and three types τ_1, τ_2, τ_3 is a property of statements and expressions. By $(m, \tau_1, \tau_2, \tau_3) \models s$ (shortly: $\models s$) we denote that s is correct wrt. $m, \tau_1, \tau_2, \tau_3$. This is defined by:

- $\sigma \in \text{statement}^* \wedge \forall n \in \text{dom } \sigma : \models \sigma(n) \Rightarrow \models \sigma$
- $\models e \Rightarrow \models (\mathbf{expr}, e)$
- $\models \mathbf{null}$
- $\models \mathbf{nil}$
- $x \in \text{identifier} \wedge x \in \text{dom } \delta_m \wedge \delta_m(x) \in (\text{constant} \cup \mathbf{a_constant}) \Rightarrow \models x$
- $\models \mathbf{loc}$
- $\models \mathbf{par}$
- $\tau \in \text{type} \wedge \forall x \in \Xi(\tau) : x \in \text{dom } \delta_m \wedge \delta_m(x) \in \text{type} \cup \{\mathbf{type}\} \Rightarrow \models (\mathbf{new}, \tau)$
- $\models e \wedge e \vdash (\mathbf{var}, \tau) \Rightarrow \models (\mathbf{deref}, e)$
- $\models e \wedge e \vdash \tau \wedge \models e' \wedge (e', q) \in \text{target} \wedge e' \vdash \tau' \wedge \tau' \dashv q \wedge (e', q) \vdash (\mathbf{var}, \hat{\tau}) \wedge \tau \lesssim_m \hat{\tau} \Rightarrow \models (\mathbf{assign}, e, (e', q))$
- $e \in \text{expression}^+ \wedge \forall n \in \text{dom } e : \models e(n) \Rightarrow \models (\mathbf{tuple}, e)$
- $e \in \text{expression}^+ \wedge \exists \tau \forall n \in \text{dom } e : (\models e(n) \wedge e(n) \vdash \tau) \Rightarrow \models (\mathbf{array}, e)$
- $e \in \text{expression}^+ \wedge \exists \tau \forall n \in \text{dom } e : (\models e(n) \wedge e(n) \vdash \tau) \Rightarrow \models (\mathbf{list}, e)$

- $f \in (index \times expression)^+ \wedge (\forall n \in dom f : f(n) = (x, e) \Rightarrow \models e) \wedge$
 $(\forall n, n' \in dom f : (f(n) = (x, e) \wedge f(n') = (x, e')) \Rightarrow n = n')$
 $\Rightarrow \models (struct, f)$
- $\models e \wedge e \vdash (struct, \tau) \wedge x \in dom \tau \Rightarrow \models (select, e, x)$
- $\models e \wedge e \vdash (tuple, \tau) \wedge n \in dom \tau \Rightarrow \models (select, e, n)$
- $\models e \wedge e \vdash (array, n, \tau) \wedge e' \in expression \wedge \models e' \Rightarrow \models (select, e, e')$
- $\models e \wedge e \vdash (list, \tau) \wedge e' \in expression \wedge \models e' \Rightarrow \models (select, e, e')$
- $\models e \wedge e \vdash (rout, \tau, \tau') \wedge \models e' \wedge e' \vdash \tau'' \wedge \tau'' \lesssim_m \tau \Rightarrow \models (apply, e, e')$
- $\models e \wedge \gamma \in (statement^*)^+ \wedge \forall n \in dom \gamma : \models \gamma(n) \Rightarrow \models (switch, e, \gamma)$
- $\sigma \in statement^* \wedge \models \sigma \Rightarrow \models (loop, \sigma)$
- $n \in number \Rightarrow \models (break, n)$
- $\models e \wedge e \vdash \tau \wedge \tau \lesssim_m \tau_3 \Rightarrow \models (return, e)$

5.3 Semantics

For any program_system Π the function $\llbracket \cdot \rrbracket : literal \rightarrow U$ assigns a certain value $\llbracket l \rrbracket$ of universe U to a literal l . For the description of $\llbracket \cdot \rrbracket$ we need the sets

- $\Phi := \bigcup \{ [V_\tau \rightarrow \tau] \mid \tau \in type \} \cup \{ \underline{nil}, \emptyset \}$
- $SC := \{ \uparrow, \downarrow \} \cup \{ \downarrow_i \mid i \in \mathbb{N} \}$
- $V := \bigcup \{ V_\tau \mid \tau \in types \}$.

Function $\llbracket \cdot \rrbracket$ has to fulfill the following conditions:

- $\llbracket null \rrbracket = \emptyset$
- $\llbracket nil \rrbracket = \underline{nil}$
- $x \in defined_constant \wedge \delta_\Pi(x) = (\tau, l) \Rightarrow \llbracket x \rrbracket = \llbracket l \rrbracket$
- $l \in basic_literal \Rightarrow \llbracket l \rrbracket \in U_B$
- $\llbracket anonymous \rrbracket \in V$
- $\lambda \in literal^+ \Rightarrow \llbracket \lambda \rrbracket \in U^+$
 where $dom \lambda = dom \llbracket \lambda \rrbracket \wedge \forall n \in dom \llbracket \lambda \rrbracket : \llbracket \lambda \rrbracket(n) = \llbracket \lambda(n) \rrbracket$
- $\lambda \in [index \rightarrow literal] \Rightarrow \llbracket \lambda \rrbracket \in [index \rightarrow U]$
 where $dom \lambda = dom \llbracket \lambda \rrbracket \wedge \forall x \in dom \llbracket \lambda \rrbracket : \llbracket \lambda \rrbracket(x) = \llbracket \lambda(x) \rrbracket$
- $(\tau, b) \in routine_literal \Rightarrow \llbracket (\tau, b) \rrbracket : \Phi \times U \rightarrow \Phi \times U$
 where

$$\llbracket (\tau, b) \rrbracket(\varphi, p) = \begin{cases} (\varphi'', p') & \text{if } \llbracket b \rrbracket(\varphi', p, l, \emptyset, \uparrow) = (\varphi'', p, l, p', \downarrow) \text{ for some } l \in \llbracket \tau \rrbracket \\ & \wedge dom \varphi' = dom \varphi \cup \{l\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $\sigma \in statement^* \Rightarrow \llbracket \sigma \rrbracket : \Phi \times U \times V \times U \times SC \rightarrow \Phi \times U \times V \times U \times SC$
 where
 - $\sigma = \emptyset \vee x \neq \uparrow \Rightarrow \llbracket \sigma \rrbracket(\varphi, p, l, v, x) = (\varphi, p, l, v, x)$ and else
 - $s \in statement \Rightarrow \llbracket s.\sigma \rrbracket = \llbracket s \rrbracket \llbracket \sigma \rrbracket$.

- $s \in \text{statement} \Rightarrow \llbracket s \rrbracket : \Phi \times U \times V \times U \times SC \rightarrow \Phi \times U \times V \times U \times SC$

where

- $e \in \text{expression} \wedge \llbracket e \rrbracket(\varphi, p, l) = (\varphi', u)$
 $\Rightarrow \llbracket (\text{expr}, e) \rrbracket(\varphi, p, l, v, x) = (\varphi', p, l, u, x)$

- $e \in \text{expression} \wedge \gamma \in (\text{statement}^*)^+ \wedge \llbracket e \rrbracket(\varphi, p, l) = (\varphi', u)$

$$\Rightarrow \llbracket \text{switch}, e, \gamma \rrbracket(\varphi, p, l, v, x) = \begin{cases} \llbracket \gamma(u) \rrbracket(\varphi', p, l, v, x) & \text{if } u \in \text{dom } \gamma \\ \llbracket \gamma(|\gamma|) \rrbracket(\varphi', p, l, v, x) & \text{otherwise} \end{cases}$$

- $\sigma \in \text{statement}^*$

$$\Rightarrow \llbracket (\text{loop}, \sigma) \rrbracket(\varphi, p, l, v, x) = \begin{cases} (\varphi, p, l, v, \Downarrow) & \text{if } x = \Downarrow \\ (\varphi, p, l, v, \downarrow_n) & \text{if } x = \downarrow_{n+1} \\ (\varphi, p, l, v, \uparrow) & \text{if } x = \downarrow_0 \\ (\llbracket \sigma \rrbracket \llbracket (\text{loop}, \sigma) \rrbracket)(\varphi, p, l, v, x) & \text{otherwise} \end{cases}$$

- $n \in \text{number}$

$$\Rightarrow \llbracket (\text{break}, n) \rrbracket(\varphi, p, l, v, x) = (\varphi, p, l, v, \downarrow_n)$$

- $e \in \text{expression} \wedge \llbracket e \rrbracket(\varphi, p, l) = (\varphi', u)$

$$\Rightarrow \llbracket (\text{return}, e) \rrbracket(\varphi, p, l, v, x) = (\varphi', p, l, u, \Downarrow)$$

- $x \in \text{index} \cup \text{number} \Rightarrow \llbracket x \rrbracket : \Phi \times U \times V \rightarrow \Phi \times (\text{index} \cup \text{number})$

where $\llbracket x \rrbracket(\varphi, p, l) = (\varphi, x)$

- $q \in \text{qualification} \Rightarrow \llbracket q \rrbracket : \Phi \times U \times V \rightarrow \Phi \times (\text{index} \cup \text{number} \cup U)^*$

where

$$\llbracket \emptyset \rrbracket(\varphi, p, l) = (\varphi, \emptyset) \text{ and}$$

$$\llbracket x.q \rrbracket(\varphi, p, l) = (\varphi'', y.\psi) \text{ if } \llbracket x \rrbracket(\varphi, p, l) = (\varphi', y) \text{ and } \llbracket q \rrbracket(\varphi', p, l) = (\varphi'', \psi)$$

- $e \in \text{expression} \Rightarrow \llbracket e \rrbracket : \Phi \times U \times V \rightarrow \Phi \times U$

where

- $\llbracket \text{loc} \rrbracket(\varphi, p, l) = (\varphi, l)$

- $\llbracket \text{par} \rrbracket(\varphi, p, l) = (\varphi, p)$

- $\llbracket \text{null} \rrbracket(\varphi, p, l) = (\varphi, \emptyset)$

- $\llbracket \text{nil} \rrbracket(\varphi, p, l) = (\varphi, \underline{\text{nil}})$

- $x \in \text{identifier} \wedge \delta_{\Pi}(x) = (\tau, c)$

$$\Rightarrow \llbracket x \rrbracket(\varphi, p, l) = (\varphi, \llbracket c \rrbracket)$$

- $\tau \in \text{type} \wedge v \in V_{\tau} \wedge v \notin \text{dom } \varphi \wedge u \in \llbracket \tau \rrbracket \wedge \varphi' = \varphi \cup \{(v, u)\}$

$$\Rightarrow \llbracket (\text{new}, \tau) \rrbracket(\varphi, p, l) = (\varphi', v)$$

- $e \in \text{expression} \wedge \llbracket e \rrbracket(\varphi, p, l) = (\varphi', v)$

$$\Rightarrow \llbracket (\text{deref}, e) \rrbracket(\varphi, p, l) = (\varphi', \varphi'(v))$$

- $q \in \text{qualification} \wedge e \in \text{expression} \wedge e' \in \text{expression}$

$$\llbracket q \rrbracket(\varphi, p, l) = (\varphi', \psi) \wedge \llbracket e' \rrbracket(\varphi', p, l) = (\varphi'', v) \wedge \llbracket e \rrbracket(\varphi'', p, l) = (\hat{\varphi}, u)$$

$$\Rightarrow \llbracket (\text{assign}, e, (e', q)) \rrbracket(\varphi, p, l) = (\hat{\varphi}[v.\psi/u], u)$$

- $e \in \text{expression}^+ \wedge u : \text{dom } e \rightarrow U \wedge$

$$\forall n \in \text{dom } e : \llbracket e(n) \rrbracket(\varphi_{n-1}, p, l) = (\varphi_n, u(n))$$

$$\Rightarrow \llbracket (\text{tuple}, e) \rrbracket(\varphi_0, p, l) =$$

$$\llbracket (\text{array}, e) \rrbracket(\varphi_0, p, l) =$$

$$\llbracket (\text{list}, e) \rrbracket(\varphi_0, p, l) = (\varphi_{|e|}, u)$$

- $f \in (\text{index} \times \text{expression})^+ \wedge u : \{x_1, \dots, x_{|f|}\} \rightarrow U \wedge$
 $\forall n \in \text{dom } f : f(n) = (x_n, e_n) \wedge \llbracket e_n \rrbracket(\varphi_{n-1}, p, l) = (\varphi_n, u(x_n))$
 $\Rightarrow \llbracket (\text{struct}, f) \rrbracket(\varphi_0, p, l) = (\varphi_{|f|}, u)$
- $e \in \text{expression} \wedge x \in (\text{index} \cup \text{number} \cup \text{expression}) \wedge$
 $\llbracket x \rrbracket(\varphi, p, l) = (\varphi', v) \wedge \llbracket e \rrbracket(\varphi', p, l) = (\varphi'', u)$
 $\Rightarrow \llbracket (\text{select}, e, x) \rrbracket(\varphi, p, l) = (\varphi'', u(v))$
- $e, e' \in \text{expression} \wedge \llbracket e \rrbracket(\varphi, p, l) = (\varphi', u) \wedge \llbracket e' \rrbracket(\varphi', p, l) = (\varphi'', v) \wedge u(\varphi'', v) = (\hat{\varphi}, w)$
 $\Rightarrow \llbracket (\text{apply}, e, e') \rrbracket(\varphi, p, l) = (\hat{\varphi}, w)$

5.4 Constraints

Let Π be any `program_system`. Then,

- \Subset must be irreflexive

and, for each module $m \in \Pi$ it must hold:

- $x \in \text{dom } \delta_m \wedge \delta_m(x) = (\tau, l) \in \text{constant} \wedge y \in \Xi(l)$
 $\Rightarrow y \in \text{dom } \delta_m \wedge \delta_m(y) \in (\text{constant} \cup \text{a_constant})$

5.5 Explanations

Literals describe values of constants. Of course, a value of a constant has to be member of the meaning of its type. Therefore, a literal used in a constant must be compatible to its type. For a `defined_constant`, i.e. an identifier x with $\delta_\Pi(x) = (\tau, l)$, x is by definition compatible to τ .

There are two predefined constants: `null` and `nil`. The meaning of `null` is the emptyset which represents an undefined value. `null` is compatible to the type `void`. The meaning of `nil` is the special variable `nil`. The value of it is undefined. `nil` is compatible to any `variable_type`.

The definition of `basic_literals` which describe values of `basic_types` is left open since also `basic_types` are still undefined.

Values of variables are generated by the `program_system` itself. Therefore, the user do not need to describe them. Here, the special literal `anonymous` describes this situation.

Literals can be combined to a nonempty sequence (literal^+) and to a function ($[\text{index} \rightarrow \text{literal}]$) in order to describe values of `tuple_types`, `array_types`, `list_types` and `struct_types`. This composition must be compatible to the corresponding types, that means the lenght of the sequence must coincide with the number of components of the type and every literal of the sequence has to be compatible to the type of the corresponding component.

The meaning of a `routine_literal` is a partial function from $\Phi \times U$ to $\Phi \times U$. Here, Φ is the set of assignments of values to variables and U is the universe of all values. If (τ, b) is a `routine_literal` and $f : \Phi \times U \rightarrow \Phi \times U$ is a function of its meaning $\llbracket (\tau, b) \rrbracket$ then $f(\varphi, p) = (\varphi', p')$ provided that $(\varphi, p) \in \text{dom } f$. That means, function f is applied onto the current variable-assignment φ and a parameter p . If it is defined then a possibly modified variable-assignment φ' and a value p' results. The modification of variable-assignment φ to φ' is called the side-effect of f .

The type τ has to be a `variable_type` and describes the `local_variable`. This `local_variable` exists only during the run of the routine. It is generated dynamically when the routine is entered and deleted after the exit. A `local_variable` allows to assign intermediate values of the routine without use of the parameter of the routine or of a variable which is a constant of the `program_system`. This is especially helpful for recursive routines. Note, that only one `local_variable` exists. If more then one value should be stored locally then the type of the `local_variable` must be structured. For instance, by `(var, (list, τ))` any number of values, all of type τ , can be stored locally as a list. If no local values are needed then the type of the `local_variable` may be choosen as `void`, more exactly: the `local_variable` is then defined as `(var, void)`.

The body of a routine is just a sequence σ of statements. Its meaning is a partial function $\llbracket \sigma \rrbracket : \Phi \times U \times V \times U \times SC \dashrightarrow \Phi \times U \times V \times U \times SC$. If $\llbracket \sigma \rrbracket(\varphi, p, l, v, x) = (\varphi', p, l, x')$ then

- φ, φ' are assignments of values to variables.
- p is the value of the actual parameter onto which the routine is applied. This value is never changed during the run of the routine. Note, that only one parameter exists. If more values should be passed as parameters then they have to be composed to one. In this case, parameter p has a composed type (`tuple_type`, `array_type`, `list_type` or `struct_type`).
- l is the local variable of the routine which is also never changed during the run of the routine.
- v is a local value generated by expressions.
- x indicates the interruption of statement sequence. While $x = \uparrow$ the statement sequence has to be evaluated statement by statement. Otherwise, the evaluation of the statement sequence is interrupted and the rest is ignored.

Apart from syntax, there are restrictions for a correct construction of a body. These are defined by soundness and correctness of statements and expressions. As an auxiliary notion, type inference rules allow to infer types from expressions.

For a routine literal (τ, b) , it holds $\llbracket (\tau, b) \rrbracket(\varphi, p) = (\varphi'', u)$ if and only if $\llbracket b \rrbracket(\varphi', p, l, \emptyset, \uparrow) = (\varphi'', p, l, \downarrow)$. The type τ has to be similar to a `variable_type` (`var`, τ'). As mentioned above, l denotes any variable from $V_{\tau'}$. φ' equals φ extended by a pair (l, w) where w is any value from $\llbracket \tau' \rrbracket$. That means:

$$\varphi'(v) = \begin{cases} w & \text{if } v = l \\ \varphi(v) & \text{otherwise} \end{cases}$$

The meaning of a statement is a partial function of the same kind as the meaning of a statement sequence.

An `exp_statement` has the form (expr, e) where e is an expression. The meaning of an expression e is a partial function $\llbracket e \rrbracket : \Phi \times U \times V \dashrightarrow \Phi \times U$. For the evaluation of an `exp_statement` we get $\llbracket (\text{expr}, e) \rrbracket(\varphi, p, l, v, x) = (\varphi', p, l, u, x)$ if $\llbracket e \rrbracket(\varphi, p, l) = (\varphi', u)$.

The evaluation of a `switch_statement` $(\text{switch}, e, \gamma)$ where e is an expression and γ is a nonempty sequence of sequences of statements is done in two steps.

- At first, the expression e is evaluated where we get a, possibly, modified variable-assignment φ' and a value u .
- Secondly, if the u -th element of sequence γ exists then the element $\gamma(u)$ which is a sequence of statements is evaluated. Otherwise, the last element of γ is evaluated.

In this way, the last element of sequence γ is the default element which is always evaluated if the value u does not fit with any index of γ . If one wants that the default element is without any effect then this element must be the empty sequence. The same holds for other elements.

The evaluation of a `loop_statement` (loop, σ) where σ is a sequence of statements is the repeated evaluation of sequence σ . This repetition is interrupted as soon as the sign x becomes \downarrow , or \downarrow_n . In the first case, \downarrow is not influenced, in the latter, n is decreased by one or, if $n = 0$ then \downarrow_0 is changed to \uparrow . Therefore, \downarrow_n causes to interrupt all n enclosed loops at once.

The evaluation of a `break_statement` (break, n) where n is a number sets the indicator x to \downarrow_n which, as it is mentioned before, causes the interruption of the n enclosed loop-statements. Note, however, if at the position of the `break_statement` in the body of a routine there are only less than n loops enclosed then an abnormal termination of the routine happens and the result is undefined!

The evaluation of a `return_statement` (return, e) where e is an expression is done in two steps.

- At first, the expression e is evaluated where we get a variable-assignment φ' and a value u .
- Secondly, the tuple $(\varphi', p, l, u, \downarrow)$ is taken as the result. That means, the indicator x is set to \downarrow which causes a normal termination of the whole body of the routine.

Since for an expression e , $\llbracket e \rrbracket(\varphi, p, l) = (\varphi', u)$, the evaluation of an expression may change the variable-assignments φ' as a side-effect and generates a value u .

If the expression is one of the atoms `loc`, `par`, `null`, `nil` or an identifier x then φ is not changed. Then, the value u equals to

- l , the local variable, if the expression is `loc`,
- p , the parameter, if the expression is `par`,
- \emptyset , the emptyset, if the expression is `null`,
- `nil`, if the expression is `nil`,
- $\llbracket c \rrbracket$, the meaning of the constant c , if the expression is an identifier x and the definition of x is (τ, c) .

If the expression is a generation (`new`, τ) then we have:

- The new variable-assignment φ' is got by φ , the old one, extended by one pair (v, w) where v is any variable not contained already in the domain of φ and w is any value of type τ . That means, a new variable v is generated and its assigned value is not determined.
- The value u equals to the generated variable v .

If the expression is a dereferencing (`deref`, e) where e is an expression then we have:

- At first, e is evaluated where we get the pair (φ', v) . φ' is taken as the new variable-assignment. If (`deref`, e) is correct then the inferred type of e has to be similar to a variable_type (`var`, τ). Therefore, the value v is a variable.
- As the value u of the whole expression the assigned value $\varphi'(v)$ is taken. That means, we go over from a variable to its assigned value.

If the expression is an assignment (`assign`, e , (e', q)), where (e', q) is a target, then the evaluation is a bit more complicated. As the main constraint, the inferred type of expression e has to be a subtype of the inferred type of the target (e', q) .

- At first, the qualification q is evaluated where we get a variable-assignment φ' and a sequence ψ which elements are indices, numbers and/or values. The details about this evaluation are described below.
- Secondly, expression e' is evaluated where we get $\llbracket e' \rrbracket(\varphi', p, l) = (\varphi'', v)$. The value v is a variable.
- Thirdly, expression e is evaluated where we get $\llbracket e \rrbracket(\varphi'', p, l) = (\hat{\varphi}, u)$.
- Finally, the variable-assignment $\hat{\varphi}$ is modified in such a way that value u becomes the new component of the value $\hat{\varphi}(v)$ at position ψ . That means, if $\hat{\varphi}'$ is the resulting variable-assignment then $\hat{\varphi}'(v, \psi) = \hat{\varphi}'(v)(\psi) = u$. The resulting value is u .

A qualification q is a sequence which elements may be indices, numbers or expressions. Its meaning is a partial function $\llbracket q \rrbracket : \Phi \times U \times V \dashrightarrow \Phi \times (\text{index} \cup \text{number} \cup U)^*$. It is evaluated from the left to the right, element by element. The resulting value of step k is

- $q(k)$, if the element $q(k)$ is an index or a number and
- u , if $q(k)$ is an expression and u is the value got by the evaluation of $q(k)$,
i.e. $\llbracket q(k) \rrbracket(\varphi_{k-1}, p, l) = (\varphi_k, u)$.

The resulting values of every step are combined to the resulting sequence $\psi \in (\text{index} \cup \text{number} \cup U)^*$. Note, that the variable-assignment is not changed by the evaluation of an index and a number!

If the expression is a construction then there exist two main cases:

- The construction has one of the forms (`tuple`, e), (`array`, e), (`list`, e) where e is a nonempty sequence of expressions. In this case, sequence e is evaluated from the left to the right, element by element. All the resulting values are combined to a tuple, an array, or a list respectively.

- The construction has the form **(struct, f)** where f is a nonempty sequence of pairs $f(n) = (x_n, e_n)$ with $n = 1, \dots, m$ (m is the length of sequence f). Here, x_n is an index and e_n is an expression. In this case, the expressions e_1, \dots, e_m are evaluated from the left to the right where we get the values u_1, \dots, u_m . Then, a function $u : \{x_1, \dots, x_m\} \rightarrow U$ with $u(x_n) = u_n$ results provided that the evaluation of all the expressions terminates.

If the expression is a selection (**select, e, x**) where e is an expression and x is an index, a number or an expression then evaluation is done in the following way:

- At first, expression e is evaluated. As resulting value we get a function u which is a value composed by several components.
- Secondly, x is evaluated. Let v be the result of this step.
- The finally resulting value is $u(v)$, the component v of u .

If the expression is an application (**apply, e, e'**) where e and e' are expressions then the evaluation is also done in three steps.

- At first, expression e is evaluated where we get a variable-assignment φ' and a value u .
- Secondly, expression e' is evaluated where we get a variable-assignment φ'' and a value v .
- If (**apply, e, e'**) is correctly built then the inferred type of e is similar to **(rout, τ, τ')** and for the inferred type τ'' of e' $\tau'' \lesssim_m \tau$ holds. Therefore, value u is a partial function from $[\Phi \times T \rightarrow \Phi \times T']$ where $[[\tau'']] \subseteq [[\tau]] \subseteq T \subseteq U$ and $T' \subseteq [[\tau']] \subseteq U$ and value v is an element from $[[\tau'']]$, consequently: $v \in T$.

Finally, these facts allow to apply function u onto φ'' and v . If this application is defined then we get $u(\varphi'', v) = (\hat{\varphi}, w)$ where $w \in [[\tau']]$. The pair $(\hat{\varphi}, w)$ is the result of the evaluation of **(apply, e, e')**.

6 Conclusions

The here described structure of the database for a programming information system is a first version. It is the basis to design the interface and the manager. Implementing these, a new or modified understanding may arise. This can drive us to change the concepts.

On the other hand, programming concepts are not strictly given in an objective manner. They depend also on the designer as a matter of taste. So, alternative concepts exist which, however, would influence the structure of the database a lot. For instance, types could be values. As a consequence, dynamic type checking is necessary if the type of an expression cannot be inferred statically. Therefore, we did not allow this in our approach.

Some of the here introduced concepts may be generalized. Two of them are favourite candidates:

- The pretype-relation may be extended by a special type construction which allows to extend explicitly a `tuple_type`, `array_type` and `struct_type` by new components. This would correspond to the construction of a class on the basis of another one in object-oriented languages like Smalltalk, C++, Java, ...
- The hiding of types in the export could be generalized in the way that only components of a type are hidden. This is only possible in the current approach if these components are described by type identifiers for which the details are hidden.

References

- [1] Bachmann, Peter, *Objektorientierte Programmierung unter einer (anderen) konzeptuellen Sicht*, BTU Cottbus, Reihe Informatik I-04/1998.
- [2] Van Wijngaarden, A., Mailloux, B., Peck, J., Koster, C., *Revised Report on the Algorithmic Language Algol-68*, Numerische Mathematik **14**, 2,1969, 79-218.
- [3] Wirth, N., *The programming language Oberon*, Software - Practice and Experience, **18**, 661-70.
- [4] Wirth, N. *Programming in Modula-2*, 3rd ed. Springer-Verlag, New York, 1985.