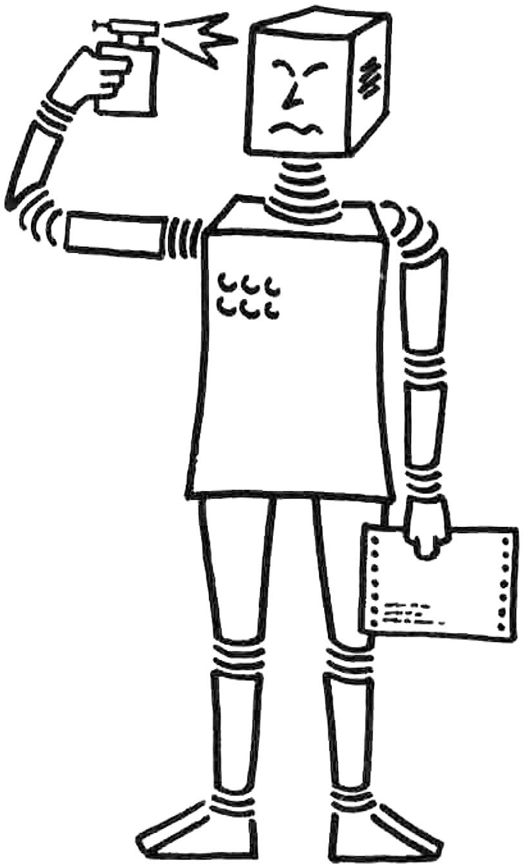
SEKI-PROJEKT

SEKI
MEMO

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



Semantics of ModPascal

Walter Olthoff

Memo SEKI-84-10

Semantics of ModPascal

Walter Olthoff

Dept. of Computer Science
University of Kaiserslautern
P.O. Box 3049
6750 Kaiserslautern
Federal Republic of Germany

Abstract

A denotational semantics is given for the programming language ModPascal, an object oriented procedural language. It employs concepts of abstract data type theory: heterogenous order algebras with strict operations describe the semantics of types and of a complete program, and the parameterization concept of ModPascal is based on explicit actualization by signature morphisms. This allows to treat standard language objects and user-defined objects in a uniform and sound way. Additionally, the semantic domain structure is able to support equivalence proofs in the transition from applicative languages to ModPascal as it is necessary in software development environments.

Keywords: Denotational Semantics. Semantics of Types. Abstract Data Types. Parameterization of Types. Software Engineering Environments.

Content:

1. Introduction

1.1. The ModPascal Environment

The procedural programming language ModPascal was developed as part of the Integrated Software Development- and Verification System (ISDV-System, [BGGORV 83]). This system employs software engineering techniques along the "verify-while-develop" paradigm: newly introduced structures are verified against formal specifications as soon as possible so that errorneous or inadequate design is detected early before it causes greater damage (=cost of system redesign). This technique is used to link the very first formal specification, the intermediate specification structures and the final ModPascal program by assigning proof tasks (correctness criteria) to all refinement steps. Then, the validity of all proof tasks implies that the ModPascal program meets the requirements imposed by the first formal specification - a proposition that is highly valuable for almost all software developments.

The applied method involves different levels of abstraction and provides concepts and tools for a verifiable transition from abstract to concrete structures. In figure 1-1 a rough overview of the various levels is given together with an also rough classification, and the verification tasks are located.

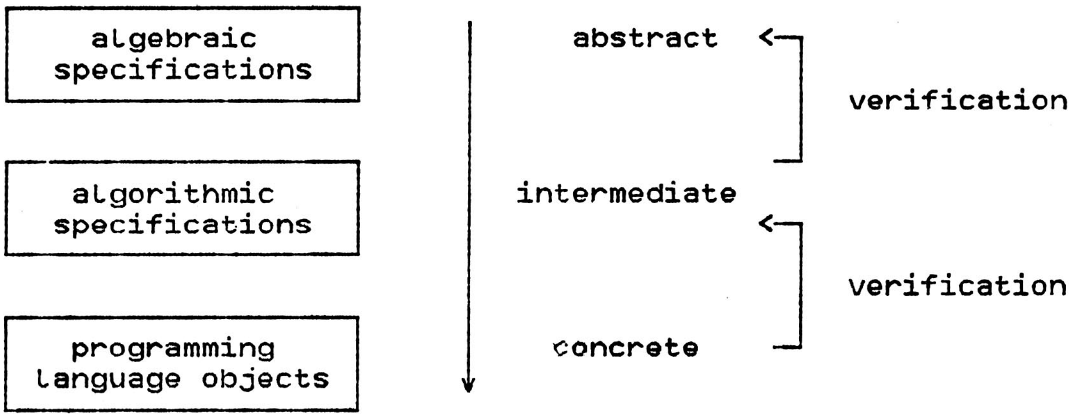


Fig. 1-1: ISDV-System scenario

The formal specifications are given in the applicative specification language ASPIK ([BV 83]) that is strongly based on algebraic specifications ([ADJ 78], [EKP 78]). ASPIK supports incremental, hierarchical software design and offers a number of powerful description features. It is the language of the 'abstract' and 'intermediate' levels of program development in the ISDV-System; the language of the 'concrete' level is ModPascal. As a consequence, both languages offer constructs that are semantically equivalent (e.g. ASPIK specifications - ModPascal modules/enrichments) but exploit the advantages of applicative/procedural languages resp.

ModPascal is an extension of Pascal [ISO 7185] in a way that preserves the full set of features of Pascal. The extension has been influenced by two facts:

- In software engineering research algebraic specifications have become widely recognized as a representation independant description method for data types (abstract data types). Algebraic specifications allow modularization and sometimes hierarchization of problem domains and they constitute referential transparency on the specification level (see e.g. [ADJ 78], [EKP 78], [GHM 78], [BV 83]).
- Existing software engineering environments still lack a satisfactory solution to fill the gap between the specification and the final programming languages (e.g. [Sil 80]). Often, it is an incompatibility of language constructs and underlaid semantics that causes the problems.

As a consequence ModPascal has been designed to meet requirements imposed by both theory of algebraic specifications and software engineering environments.

There are four new kinds of objects that make ModPascal differ from Pascal: modules, enrichments, instantiations and instantiate types. The term 'type' in its usual (Pascal) sense is not applicable to the first three of these constructs since they model more or different information than array, record etc.

For an intuitive introduction to the new concepts see [Olt 84], section 1. In dependence of this modifications and enlargements of the Pascal type set, also modifications on variable declarations, assignments, and operation calls are necessary, and their semantics as well as the semantics of the objects have to be defined.

The language definition of ModPascal is divided into two levels: syntax and static semantics of all additional constructs to Standard Pascal are given in [Olt 84], including those portions of Standard Pascal that are mandatory for every semantic description (e.g. assignments, operation calls); the dynamic semantics is defined in this paper.

We use the technique of denotational semantics. Then, the new idea is to provide an appropriate domain of algebras as target of the meaning assignment for object definitions (types, modules, enrichments), such that operations and value sets of a structure are combined. In this setting, the Pascal predefined types as well as the types generated by type constructors as array, record are given algebras as meanings. The description of the semantics is focussed onto this embedding problem of algebra domains. Other necessary topics of a complete and detailed language semantics (e.g. block structuring, jumps) are suppressed, since adequate description techniques are well-known (environment changes, continuations), and their involvement would considerably increase complexity of semantic clauses.

Another important domain provides the meaning for instantiation objects, a kind of signature morphisms (see sec. 2.2.1.) in concrete procedural languages. Instantiations can be viewed at as mappings between identifiers, and their main purpose is to realize the object parameterization concept of ModPascal (see sections 3.6. and 3.7.). There, it is necessary to express the connection between 'formal' and 'actual' parameter objects in form of instantiation definitions, and their semantics is captured by the introduction of a domain of special identifier mappings.

For a convieniant description of the ModPascal semantics, here the grammar of [Olt 84] is reformulated in Vienna Definition Language (VDL, [Weg 72]). Upon this, the definition of the syntactic domains is based. Syntactic domains, semantic domains, and semantic functions are introduced in section 2. Section 3 defines the meaning of all ModPascal-specific constructs and some Standard Pascal constructs. Section 4 illuminates the semantical questions arising from the fact that the verification context mentioned above makes it necessary to precompile portions of ModPascal code to Pascal.

1.2. Notations

N denotes the set of natural numbers.

For a natural number n, (n) denotes the set {1, ..., n}, and [n] := (n) $\cup$ {0}.

For vectors v = $(v_1, ..., v_n)$, $(v_1, ..., v_n)\downarrow i$ or $v\downarrow i$ denotes the i-th component v_i of v.

For a set s, $\mathcal{P}$(s) denotes the power set of s.

$\hat{\exists}$ denotes the unique existential quantification.

For a mapping m: A —> B defined by m $\subseteq$ (A x B), the <u>substitution</u> m[a $\leftarrow$ a_1] denotes (m \ {(a, m(a))}) $\cup$ {(a, a_1)}.

Four operators are used for functional abstraction:

- λx . term: Bounds free occurrences of x in term. This abstraction is equivalent to a definition: F(x) = term of a function F
- ιx . cond : Bounds x in cond and qualifies the x as unique to fullfill cond. Equivalent to: $\hat{\exists}$ x . (cond = true). If no unique x exists, ι evaluates to $\perp$. Example: n := ιi . (i+1=5) $\Rightarrow$ (n=4)
- fix f . term: Bounds free occurrences of f in term and denotes the least fixpoint of the functional equation F = term[F] where term[F] is a term with free occurrences of F.

Example: fix f . (λn . if n = 0 then 1 else n*f(n-1))
denotes the least fixpoint of the functional equation F(n) = if n = 0 then 1 else n*F(n-1), that is the standard faculty function.

- η x . cond :
  bounds x in cond and qualifies x as one possible value that satisfies cond. Equivalent to: $\exists$ x . (cond = true). If no value exists that satisfies cond, η evaluates to $\perp$.
  Example: n := η x . (x*x = 9) $\Rightarrow$ n $\in$ {3, -3})

If indexed items occur themselves in index positions, the indices are juxtaposed in parenthesis.
Example: $X_n \longrightarrow Y_{X(n)} \longrightarrow Z_{Y(X(n))}$
$X_{ij} \longrightarrow Y_{X(i)(j)}$

2. Domains

To state the semantics of ModPascal we employ the technique of denotational semantics ([Sto 77], [Gor 79]). We have to define syntactic domains, semantic domains and functions mapping syntactic constructs to their meaning in a semantic domain.

2.1. Syntactic Domains

A convenient way to describe the syntax of ModPascal is by Vienna Definition Language (VDL, [Weg 72]). We briefly sketch some basic concepts that are important for our purposes, and then state the ModPascal grammar of [Olt 84] in VDL.

2.1.1. VDL

VDL supports the idea of abstract syntax in that sense, that no familiar language symbols as 'begin' or 'end' (i.e. the terminal vocabulary) occur in a VDL description. Instead, all objects (syntactic entities) are collected in sets, and there are selectors that allow manipulation of them. Objects are separated into two kinds:

- elemantary objects: objects with no components and therefore no selectors,
- composite objects : objects which may be composed of objects by construction operators. The components may be elementary or composite objects, and each is selectable by a unique selector.

Notation: {o_1, o_2} denotes a set of elementary objects.
(s_1: C_1, s_2: C_2) denotes a set of composite objects with selectors s_1, s_2 and component object sets C_1, C_2.

Composite objects represent tree structures in which the arcs are labelled by selectors, the leaf nodes are elementary objects and all other nodes are composite objects.

There is a distinguished elementary object, the so-called <u>null object</u> $\perp$ which is different from every other elementary object. The null object is used to denote empty domains or erroneous manipulations on domains.

<u>Def. 2.1.1.-1</u> [selector application]
Let $C = (s_1: C_1, \dots, s_n: C_n)$ denote a composite object. Let s denote a selector, and let $c \in C$ with $c = (c_1, \dots, c_n)$.
Then (s c) is called <u>selector application</u> with

$$(s\ c) := \underline{\text{case}}\ s = s_i,\ i \in (n) : c_i \quad \underline{\text{otherwise}}\ \perp$$

¤

<u>Notation</u>: $(s^n\ c) := (s\ (s\ (s\ \dots\ (s\ c)\ \dots\ )$ [n times, n > 0]
$(s^0\ c) := c$

Selectors may be composed, too. If $(s_1: C_1)$ and $C_1 \equiv (s_2: C_2, s_3: C_3)$ are composite objects then $s_3 s_1$ is a <u>composite selector</u>. If $x \in (s_1: C_1)$ then $s_3 s_1$ can be applied to x to select the c_3-component.

<u>Notation</u>: If $s_n s_{n-1} \dots s_1$ denotes a composite selector, then $(s_n\ (s_{n-1}\ (\ \dots\ (s_1\ x)\ \dots)$ denotes the application to a composite object x.

<u>Def. 2.1.1.-2</u>. [admissability]
Let $s := s_n \dots s_1$ denote a composite selector, C a set of composite objects.

1) The application of s to $c \in C$, i.e.
   $(s_n\ (s_{n-1}\ (\ \dots (s_1\ c)\ \dots)$
   is <u>admissable</u>, if
   $\forall\ i \in (n)\ .\ s_i\ (s_{i-1}\ (\ \dots (s_1\ c)\ \dots) \neq \perp$.
   s is also called <u>admissable selector for c</u>.
2) AD(c) := {s| s is admissable selector for c}

¤

The following conventions and operators are used:

1) We assume all object sets to be flat domains (see definition 2.2.1.-3).

2) Syntactic Domains are denoted by identifiers starting with capital letter. Selectors and syntactic domains may occur postfixed by 'L' (for 'list'). This implies the following list structure:

   DomainL = (first: Domain, rest: DomainL)

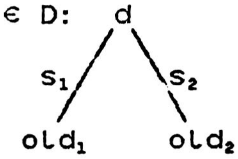
   If Domain = Dom1 $\vee$ Dom2
   then DomainL = Dom1L $\vee$ Dom2L

   An operator Length: DomainL ⟶ $\mathbb{N}$ that returns the number of list elements is defined for every domain. Length($\perp$) = 0.

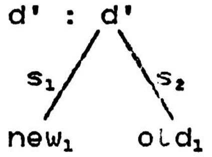
3) The L-version of a domain is not expliciteLy mentioned in the abstract syntax of ModPascal.

Special case: DomLL = (first: DomainL, rest: DomainLL)

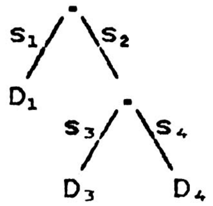
4) The general assignment operator is μ:
For d ∈ D:



$\mu(d;\ s_1:\ new_1)$:= d' :



5) The general construction operator is μ_0:
$\mu_0(s_1:\ D_1,\ s_2:\mu_0(s_3:\ D_3,\ s_4:\ D_4))$ describes the domain:



2.1.2. Abstract Syntax of ModPascal

```
Program      = (prog_head: Prog_head, block: Block)
Prog_head    = (prog_id: Id, prog_params: IdL)
ID           = {alphanumeric strings}
Block        = (labL:LabL, constL: ConstL, objL: ObjL,
                varL:VarL, sub_progL: Sub_progL, stmtL: StmtL)
Lab          = {0, ..., 9999}
Const        = (const_id: Id, const_val: Const_val)
Const_val    = Id v INT v (sign: Sign, id: Id)
INT          = {integer number}
Sign         = {+, -}
Obj          = Type_def v Enrich_def v Inst_def
Type_def     = (type_id: Id, type: Type)
Type         = Id v Stand_type v Stand_type_gen v
                Non_standard_type_gen
Stand_type = {INTEGER, BOOLEAN, REAL, CHAR}
Stand_type_gen
             = Scalar_type v Subrange_type v Array_type v
                Record_type v Set_type v File_type v Pointer_type
Scalar_type
             = (idL: IdL)
Subrange_type
             = (lower: Const, upper: Const)
Array_type = (indexL: Simple_typeL, comp: Type)
Simple_type
             = Scalar_type v Subrange_type v Id
Record_type
             = (fixedL: Fixed_partL, variant_partL:
                Variant_partL)
```

```
Fixed_part = (idL: IdL, type: Type)
Variant_part
             = (tag: Tag, variantL: VariantL)
Tag          = (tagid: Id, typeid: Id) v (typeid : Id)
Variant      = (constL: ConstL, fixedL: Fixed_partL,
               variant_partL: Variant_partL)
Set_type     = (simple_type: Simple_type)
File_type    = (type: Type)
Pointer_type
             = (type: Type)
Non_standard_type_gen
             = Module_type v Instantiate_type
Module_type
             = (useL: IdL, publicL: PublicL, local: Local,
               operationL : OperationL)
Public       = Proc_head v Func_head v Init_head
Proc_head    = (proc_id: Id, paramL: ParamL)
Param        = (idL: IdL, type: ID)
Func_head    = (func_id: Id, paramL: ParamL, result: Id)
Init_head    = (init_id: Id, paramL: ParamL)
Local        = (local_typeL: Local_typeL, local_varL: VarL,
               local_operationL: Local_operationL)
Local_type   = Simple_type v Array_type v Record_type v Set_type
               v File_type v Pointer_type
Var          = (idL: IdL, type: Type, init: Init_stmt)
Init_stmt    = Term
Term         = Simple_term v Op_designator
Simple_term
             = (op_id: Op_id, act_paramL: ExprL)
Op_id        = Id v {*, /, DIV, MOD, AND, +, -, OR, =, <>, <, >,
               <=, >=, IN}
Expr         = Id v Term v S_term
S_term       = (sign: Sign, term: Term)
Op_designator
             = (var_id: Id, op_idL: IdL, act_paramL: ExprL)
Local_operation
             = Proc_head v Func_head
Operation    = Proc_spec v Func_spec v Init_spec
Proc_spec    = (proc_id: Id, body: Block)
Func_spec    = (func_id: Id, body: Block)
Init_spec    = (init_id: Id, body: Block)
Instantiate_type
             = (base_type: Id, objectL: IdL)
Enrich_def   = (enr_id: Id, useL: IdL, addL: AddL, operationL:
               OperationL)
Add          = (add_id: Id, publicL: PublicL)
Inst_def     = (inst_id: Id, useL: IdL, obj_actL: Obj_actL,
               type_actL: Type_actL, Op_actL: Op_actL)
Obj_act      = (old: Id, new: Id)
Type_act     = (old: Id, new: Id)
Op_act       = (old: Id, new: Id)
Sub_prog     = Proc_dcl v Func_dcl
Proc_dcl     = (proc_id: Id, paramL: ParamL, body: Block)
Func_dcl     = (func_id: Id, paramL: ParamL, result: Id, body:
               Block)
Stmt         = (lab: Lab, Simple_stmt: Simple_stmt) v (lab: Lab,
                struc_stmt: Struc_stmt)
Simple_stmt
              = Assg_stmt v Proc_stmt v Goto_stmt
Assg_stmt     = (assg_var: Assg_var, expr: Expr)
Assg_var      = Id v Comp_var v Ref_var v Op_designator
Comp_var      = (array_var: Id, exprL: ExprL) v Field_designator
Field_designator
              = (comp_var: Assg_var, field_id: Id) v
                (op_designator: Op_designator) v (ref_var: Id,
                field_id: Id)
Proc_stmt     = Term
Goto_stmt     = (Lab: Lab)
Struc_stmt    = StmtL v Cond_stmt v Rep_stmt v With_stmt
Cond_stmt     = (if: Expr, then: Stmt, else: Stmt) v (case_expr:
                Expr, caseL: CaseL)
Case          = (idL: IdL, stmt: Stmt)
Rep_stmt      = While v Repeat v For
While         = (while_expr: Expr, stmtL: StmtL)
Repeat        = (stmtL: StmtL, until: Expr)
For           = (for_var: Id, Lower: Expr, upper: Expr,
                direction: {UP, DOWN}, stmtL: StmtL)
With          = (idL: IdL, stmtl: StmtL)
```

2.2. Semantic Domains

2.2.1. The Domain Alg

Objects in ModPascal will be associated to algebras. To preserve applicability of denotational semantics techniques, we require special algebra domains.
Furthermore, we use a general construction from universal algebra to denote algebra domains by signatures.

Def. 2.2.1.-1 [signature]
Let OB denote a set of object names, OP a set of notation names, i.e. OB ⊆ Id, OP ⊆ Id. The tuple Σ = (OB, OP) is called signature, if

1) $\exists$ arity : OP $\longrightarrow$ (OB* x OB)
2) Let $OP_{s,t}$:= $\{op \in OP \mid arity(op) = (s,t)\}$ in
   a) $OP = \bigcup_{\substack{s \in OB^* \\ t \in OB}} OP_{s,t}$
   b) $\bigcap_{\substack{s \in OB^* \\ t \in OB}} OP_{s,t} = \emptyset$

For arity(op) = (s,t), s is called source, and t target of op. $\varepsilon \in OB^*$ denotes the empty source. ¤

Remark: The arity function assigns functionalities to operation names, i.e. if arity(op) = (s, t) with s = s_1 ... s_n, then op is name for an operation from S_1 x ... x S_n to t.

An important notion to link signatures is the signature morphism.

Def. 2.2.1.-2 [signature morphism]
Let $\Sigma_i = (OB_i, OP_i)$, $i \in \{1, 2\}$ denote signatures.
Let f: $OB_1 \longrightarrow OB_2$ and g: $OP_1 \longrightarrow OP_2$ denote mappings.
Then the tuple (f, g) is called signature morphism if

$\forall\ op \in OP_1$.
 let $(ob_1 \ldots ob_n, ob_{n+1}) := arity_1(op)$ in
  $arity_2(g(op)) = (f(ob_1) \ldots f(ob_n), f(ob_{n+1}))$

where $arity_i$ denotes the arity function of Σ_i, $i \in \{1, 2\}$.
¤

Signature morphisms become important especially for instantiation object semantics (see sec. 3.6.).

Def. 2.2.1.-3 [flat domain]
Let S denote a set. Then $(S_\perp, \sqsubseteq)$ is called a flat domain
if 1) $\perp_s \notin S$ denotes the bottom element of S.
  $S_\perp := S \cup \{\perp_s\}$
 2) $\sqsubseteq\ \subseteq (S_\perp \times S_\perp)$ is a partial order with
  $X \sqsubseteq Y :\Longleftrightarrow X = \perp_s$ or $X = Y$
¤

Notation: If no ambiguities are possible, we denote the flat domain $S_\perp$ simply as S and the bottom element $\perp_s$ as $\perp$.

Def. 2.2.1.-4 [strict]
Let $C_1, \ldots, C_m$ denote flat domains, and $n \in (m)$. A function
 f: $C_1 \times \ldots \times C_n \longrightarrow C_{n+1} \times \ldots \times C_m$
is called strict, if
$f(c_1, \ldots, c_n) = (\perp_{C_{n+1}}, \ldots, \perp_{C_m}) \Longleftrightarrow \exists\ i \in (n)\ .\ c_i = \perp_{C_i}$
¤

Remark: There is an arity operation for strict functions.
 if f: $C_1 \times \ldots \times C_n \longrightarrow C_{n+1} \times \ldots \times C_m$
 then $arity(f) \doteq (C_1 C_2 \ldots C_n, C_{n+1} C_{n+2} \ldots C_m)$.

Def. 2.2.1.-5 [order algebra]
Let C denote a non-empty set of flat domains, F a set of strict functions f: $C_1 \times \ldots \times C_n \longrightarrow C_{n+1}$ with $C_i \in C$, $i \in (n+1)$. Then (C, F) is called an order algebra.
The elements of C are called carriersets or carriers. ¤

Def. 2.2.1.-6 [Σ-algebra, interpretation]
Let $\Sigma = (OB, OP)$ denote a signature.
An order algebra A = (C, F) is called a Σ-algebra, if there are mappings
 H_1: OB $\longrightarrow$ C
 H_2: OP $\longrightarrow$ F
that associate object names to flat domains and operation names to strict functions.
The tuple (H_1, H_2) is called (Σ-signature) interpretation for A.
¤

Def. 2.2.1.-7 [naming operations]
Let $A = (C, F)$ denote an order algebra, and Id an unbound set of identifiers.
Then obname-A: $C \longrightarrow Id$ and opname-A: $F \longrightarrow Id$
associates unique names to carrier sets and operations of A.
$obnames(A) := \{obn \mid \exists\, c \in C \,.\, obn = obname\text{-}A(c)\}$
$opnames(A) := \{opn \mid \exists\, f \in F \,.\, opn = opname\text{-}A(f)\}$

¤

Def. 2.2.1.-8 [associated signature]
Let $A = (C, F)$ denote an order algebra.
Then the signature $\Sigma(A) = (OB(A), OP(A))$ defined by
1) $OB(A) := obnames(A)$
2) $OP(A) := opnames(A)$
is called the associated signature to A.

¤

Remark: Let $\Sigma = (OB, OP)$ be a signature, $A = (C, F)$ a Σ-algebra with interpretation (H_1, H_2)
Σ is the associated signature to A, if and only if
1) $\forall\, ob \in OB \,.\, ob = obname\text{-}A(H_1(ob))$
2) $\forall\, op \in OP \,.\, op = opname\text{-}A(H_2(op))$.
This is always possible by appropriate choices of obname-A and opname-A.

These syntactical operations will be used in sec. 3.6.

Def. 2.2.1.-9 [Alg[Σ], Alg]
Let $\Sigma = (OB, OP)$ denote a signature.
Then
$Alg[\Sigma] := \{A \mid A \text{ is } \Sigma\text{-algebra}\} \cup \{\bot_\Sigma\}$
$Alg := + \{Alg[\bar{\Sigma}] \mid \bar{\Sigma} \text{ is signature}\}$
(+ denotes the direct sum of domains). ¤

The definition of the domain Alg as coalesced sum of Σ-sorted algebra domains is not unproblematic. It would allow algebras that possess as carriers "the set of all sets". Since this is a well-known paradoxon-generating construction, we assume a meta-structure called universum U whose elements are sets. There are axioms that make the "set of all sets" underivable in U. Then, all carriers of elements $A \in Alg$ are assumed to be elements of U.

Def. 2.2.1.-10 [TOI]
For each algebra $A \in Alg$, the 'type-of-interest' operator TOI is defined as follows:
$TOI(A) := (\bar{C}, \bar{F})$ where
1) $\bar{C} \in C$ denotes a distinguished carrier set
2) $\bar{F} \subseteq F$ denotes all operations having $\bar{C}$ in their arities,
i.e. $\forall\, f \in \bar{F} \,.\, \text{let } (C_1 \ldots C_n, C_{n+1}) := arity(f) \text{ in}$
$\exists\, i \in (n+1) \,.\, C_i = \bar{C}$ ¤

TOI will be used to partition carriers and functions into those that are currently new defined $((\bar{C}, \bar{F}))$ and those that have already been defined $((C \setminus \{\bar{C}\}, F \setminus \bar{F}))$.

Often, the distinguished carrier set is ambiguously denoted by the algebras name, i.e. $TOI(A) = (A, F)$.

In some cases the TOI operator will evaluate to $(\perp, \perp)$ if the algebra under consideration should explicitly be characterized in this way. This does not mean that there are no carriers or functions but that none of them is qualified as 'of-interest' (see also enrichment objects, sec. 3.5.).

The next definition deals with technical operations on order algebra's.

<u>Def. 2.2.1.-11</u> [Union, Difference]
Let $\Sigma_a = (OB_a, OP_a)$, $\Sigma_b = (OB_b, OP_b)$ denote signatures.
Let $A = (C\text{-}A, F\text{-}A) \in Alg[\Sigma_a]$, $B = (C\text{-}B, F\text{-}B) \in Alg[\Sigma_b]$ and signature interpretations $H_a = (H_{a1}, H_{a2})$, $H_b = (H_{b1}, H_{b2})$.
The <u>union of A and B</u>, denoted $A \cup B$, and <u>the difference of A and B</u>, denoted by $A \setminus B$, are given by:

```
if 1) $\forall\ ob \in (OB_a \cap OB_b)\ .\ H_{a1}(ob) = H_{a2}(ob)$
   2) $\forall\ op \in (OP_a \cap OP_b)\ .\ H_{b1}(op) = H_{b2}(op)$
   3) $\forall\ c \in (C\text{-}A \cap C\text{-}B)\ ,\ \exists\ ob \in (OB_a \cup OB_b)\ .\ H_{a1}(ob) = H_{b1}(ob) = c$
   4) $\forall\ f \in (F\text{-}A \cap F\text{-}B)\ .\ \exists\ op \in (OP_a \cap OP_b)\ .\ H_{a2}(op) = H_{b2}(op) = f$
then $A \cup B := (C\text{-}A \cup C\text{-}B, F\text{-}A \cup F\text{-}B)$
else $\perp$

if (additionally) 5) $\forall\ op \in OP_a$ .
   let $(s_1 \ldots s_n, s) := arity(op)$ in
      $\neg\ (s_i \in OB_a \setminus OB_b, i \in (n)$ or $s \in OB_a \setminus OB_b)$ and
      $\neg (H_{a1}(s_i) \in C\text{-}A \setminus C\text{-}B, i \in (n)$ or $H_{a1}(s) \in C\text{-}A \setminus C\text{-}B)$
then $A \setminus B := (C\text{-}A \setminus C\text{-}B, F\text{-}A \setminus F\text{-}B)$
else $\perp$
```

If $A \cup B$ is defined, then $A \cup B \in Alg[\Sigma_a \cup \Sigma_b]$, where $\Sigma_a \cup \Sigma_b := (OB_a \cup OB_b, OP_a \cup OP_b)$ and arity-$\Sigma_a \cup \Sigma_b$ is derived from arity-Σ_a and arity-Σ_b. This holds analogously for $A \setminus B$.

¤

This definition ensures that signature interpretations behave such that union and difference of signatures and of algebras are compatible. Since the set union identifies identical carriersets, no multiple representation can occur. If, on the other hand, one is interested in multiple occurrences of special carrier set, one has to use a tagging mechanism to distinguish between set elements of different occurrences.

<u>2.2.2. Standard Algebras</u>

In ModPascal there are four predefined standard types: BOOLEAN, INTEGER, REAL, CHAR. We define standard algebras that will be associated to standard types (see sec. 3.3.1.).

<u>(1) BOOLEAN</u>

```
Signature: Σ-B := (OB-B, OP-B) with
           OB-B = {boolean}
           OP-B = {true, false, and, or, not, =, <>,
                   <=, >=, <, >}
```

```
             arity-B: OP-B ──> (OP-B* x OP-B)
             (e.g.: arity(and) = (boolean boolean, boolean)
                    arity(true) = (ε, boolean))
Then BOOL-Alg := (C-B, F-B) ∈ Alg[Σ-B] with
C-B = {B-Val},  B-Val = {TRUE, FALSE, ⊥}
F-B = {true, false, and, or, not, =, <>, <=, >=, >, <}
      where   true := TRUE
              and(x, y) := if x then y else FALSE.
               ...
               etc.

TOI(BOOL-Alg) := B-VAL
```

<u>Remarks</u>: a) The functions of F-B are ambiguously denoted by the function names of OP-B.

b) BOOL-Alg does not contain an order operation as it is obligatory for enumeration types (ModPascal/Pascal view BOOLEAN as instance of an enumeration type).

<u>(2) INTEGER</u>

```
Signature: Σ-I := (OB-I, OP-I) with
           OB-I = {integer, boolean}
           OP-I = {succ, pred, +, -, *, div, mod, abs, odd,
                   sqr, =, <>, <=, >=, <, >}
                   ∪ OP-B
           arity-I: OP-I \ OP-B ──> (OB-I* x OB-I)
           (e.g.: arity-I(succ) = (integer, integer)
                  arity-I(<>) = (integer integer, boolean)
Then INT-Alg := (C-I, F-I) ∈ Alg[Σ-I] with
C-I = {I-Val, B-Val},  I-Val = {-maxint, ..., -1,0 , 1, ...
                                +maxint, ⊥ }
F-I = {succ, pred, +, -, *, div, mod, abs, odd, sqr, =,
       <>, <=, >=, <, >}
      ∪ F-B
     where succ(x)  := if x=0 then 1 elseif x=1 then ...
           +(x, y)  := if x = 0 then y
                          elseif x > 0 then succ(+(pred(x),y))
                          else pred(+(succ(x),y))
                ...
                etc.

TOI(INT-Alg) := I-VAL
```

<u>Remarks</u>: a) The functions of F-I \ F-B are ambiguously denoted by the function names of OP-I\OP-B.

b) Maxint is the implementation dependent boundary value for integer number representation.

c) The functions of F-I \ F-B are not the familiar Integer functions since they are assumed to obey machine arithmetic rules.

d) Type conversions (coercions) are disregarded.

(3) REAL

```
Signature: Σ-R := (OB-R, OP-R) with
           OB-R = {real, integer, boolean}
           OP-R = {-, +, *, /, abs, sqrt, sqr, sin, cos,
                  arctan, exp, ln, trunc, round}
                  ∪ OP-I
           arity-R: OP-R \ OP-I ―> (OB-R* x OB-R)
           (e.g.: arity-R(sqrt) = (real, real)
                  arity-R(round) = (real, integer))
Then REAL-Alg := (C-R, F-R) ∈ Alg[Σ-R] with
C-R = {R-Val, I-Val, B-Val},  R-Val = {x| x is floating
                                               point number}
F-R = {-, +, *, /, abs, sqr, sqrt, sin, cos, arctan, exp,
      ln, trunc, round}
      ∪ F-I
      where   sqr(x) := x * x
              trunc(x) := {fractional part of floating
                           point number x}
               ...
               etc.

TOI(REAL-Alg) := R-VAL
```

Remarks: a) The functions of F-R \ F-I are ambiguously denoted by the function names of OP-R\OP-I.

b) The floating point number representation (mantisse and characteristic size) is implementation dependent.

c) The functions of F-R \ F-I are not the familiar REAL functions since they are assumed to obey floating point arithmetic rules.

(4) CHAR

```
Signature: Σ-C := (OB-C, OP-C) with
           OB-C = {char, integer, boolean}
           OP-C = {pred, succ, ord, chr, =, <>,
                  >=, <=, <, >}
                  ∪ OP-I
           arity-C: OP-C \ OP-I ―> (OB-C* x OB-C)
           (e.g.: arity-C(chr) = (char, integer)
                  arity-C(<=) = (char char, boolean))
Then CHAR-Alg := (C-C, F-C) ∈ Alg[Σ-C] with
C-C = {C-Val, I-Val, B-Val},  C-Val = {character set}
F-C = {pred, succ, ord, chr, =, <>, <=, >=, <, >}
      ∪ F-I
      where   <(x, y) := ord(x) < ord(y)
              ord(x) := if x = a then 1 elseif x = b ...
               ...
               etc.

TOI(CHAR-Alg) := C-VAL
```

Remarks: a) The functions of F-C \ F-I are ambiguously denoted

by the function names of OP-C\OP-I.
b) The character set and its order function are implementation dependend.

(5) PRE

```
Signature: Σ-P := (OB-P, OP-P) with
           OB-P = OB-B ∪ OB-I ∪ OB-R ∪ OB-C
           OP-P = OP-B ∪ OP-I ∪ OP-R ∪ OP-C
           arity-C: <combination of the arity functions of
                         the basing signatures>
Then PRE-Alg := (C-P, F-P) ∈ Alg[Σ-P] with
C-P = {B-Val, I-Val, R-Val, C-Val}
F-P = {F-B ∪ F-I ∪ F-R ∪ F-C}

TOI(PRE-Alg) := (⊥, ⊥)
```

Remarks: a) The algebra PRE-Alg is a combination of standard algebras. It is equivalent to the (algebra) union of BOOL-ALg, INT-Alg, REAL-Alg and CHAR-Alg.
b) Since no new data is introduced in PRE-Alg - the carrier sets are already defined -, there is no type-of-interest. This phenomenon is generalized in the enrichment object (see sec. 3.5.).

2.2.3. Further Domains

Usually, the semantic domains provide the mean to express the intended semantics of a programming language construct. In the case of data types and data type generators there were only few proposals in the past how to include their semantics in a denotational description ([Ten 76], [Gor 79]). This was mainly due to the fact that a well-formalized and reasonably wide accepted answer to "what is a data type" had not been given.

So our suggestion to describe data types by algebras is derived from results of abstract data type research ([ADJ 78], [EKP 78], [CIP 81]) during the last ten years. In this environment, data types are introduced by algebraic specifications, that consist of signatures (names of data sets and names of operations) and of sets of equations that define a behaviour of operations. Then a unique semantic links the specification to a single algebra, the data type.

This idea applied to the types and type generators of conventional languages requires some prerequisites:

- An appropriate domain has to be defined to serve as semantic domain of type definitions.
- If types are algebras, then the language inheritant types as BOOLEAN, INTEGER etc. cannot be looked at only as a set of values, but as structures including operations as 'and', 'or' or '+', '≤' which in general are hided in the compiler. Therefore it should be clear which operations are associated to a given type.

- A complete ModPascal program with type definitions, operation declarations, variable declarations and statement part can also be viewed at as an algebra: the type definitions represent already algebras, the operation declarations are new operations of a 'program algebra', and the variable declarations together with the statementpart may be seen as a single (unnamed) operation applicable to an initial state and resulting in a final state. The support of this main-program-algebra view is particularly important if in verification contexts programs are linked to other structures that themselves are based on algebraic semantics.

Concerning the first point above, the domain Alg was constructed as a collection of all interesting algebras. (for reason of the cautious formulation 'interesting', see the remark after definition 2.2.1.-9). The second point is covered by the explicit definitions of algebras in the preceding section and the definitions of sec. 3.3.1. and sec. 3.3.2. These structures were composed according to the Pascal standard of [ISO 7185]. The third point is dedicated to the special section 2.2.4.

If, as in our case, the language involves additional new constructs they also should fit into the algebra semantic frame. But since module types, enrichments, instantiate types and instantiations are derived structures of abstract data type theory this requirement is trivially met.

2.2.3.1. The non-Alg Domains

Although the standard types of ModPascal are modelled by algebras with the expected carriers, it is often conveniant in already very technical clauses to access directly to boolean and integer values. Therefore the domains D_BOOL and D_INT were added.

In the following only flat domains are defined.

Notation: (A ⟶ B) denotes the domain of strict monoton functions from A to B.

D_BOOL
= {true, false}: The boolean values.

D_INT
= { ..., -2, -1, 0, 1, 2, ...}: The integer values.

Id
$= \{id|\ id \in \{A, \ldots, Z, 0, \ldots, 9\}^+ \wedge first(id) \notin \{0, \ldots, 9\}\}$: Identifier are strings of letters and digits, starting with a letter.

Loc
= {an unbound domain of locations}: If locations are interpreted as main memory addresses, Loc could be seen as integer subset. But every interpretation into

distinguishable elements will work.

AlgQual
= {MAIN, BOOLEAN, INTEGER, REAL, CHAR, SCALAR, SUBRANGE, ARRAY, RECORD, FILE, SET, POINTER, MODULE, ENRICHMENT}: The algebra qualifications indicate the basing ModPascal type of an algebra. MAIN refers to the main program algebra.

ObQual
= {LAB, CONST, VAR, PROC, FUNC, INIT, INST} + AlgQual: The object qualifications indicate either the basing ModPascal feature of an item or the basing ModPascal type.

ValQual
= $\{C \mid C = TOI(A)\downarrow 1$ for $A \in Alg\}$: All carriersets of interest for algebras in Alg. ValQual may be seen as a factorization of Alg.

OpDen
= $+_{n,m \in \mathbb{N}}$ (ValQual^n —> ValQual^m): Function between n-ary and m-ary cartesian products of ValQual. A generalization of functions of algebras of Alg.

Val
= D_BOOL + D_INT + Id + Alg + ValQual + OpDen

The following domains are not necessarily flat.

Store
= (Loc —> Val): links locations and values

Env
= (Id —> (Loc x ObQual x ValQual)): Each identifier id ∈ Id is connected to a triple. The second and third components describe properties of id.

State
= Env x Store : Characterization of a state as tuple. See also the memory model in 2.2.3.2.

Trans
= (State —> State): State transformation that are induced by programming language constructs will be described with T ∈ Trans.

ETrans
= (State —> (State x Val)): Analogously Trans, but with values out of Val.

```
D_BOOL = {true, false}
D_INT = {..., -1, 0, 1, ...}
Id = {id| id ∈ {A, ..., Z, 0, ..., 9}^+ ∧ first(id) ∉
                                              {0, ..., 9}}
Alg = + {Alg[Σ]| Σ is signature}
Loc = {unbound domains of locations}
AlgQual = {MAIN, BOOLEAN, INTEGER, REAL, CHAR, SCALAR,
           SUBRANGE, ARRAY, RECORD, FILE, SET, POINTER,
           MODULE, ENRICHMENT}
ObQual = AlgQual + {LAB, PROC, FUNC, VAR, INIT}
ValQual = {C | C = TOI(A)↓1 for A ∈ Alg}
Val = D_BOOL + D_INT + Id + Alg + ValQual + OpDen
Store = Loc ——> Val
Env = Id ——> (Loc x ObQual x ValQual)
State = Env x Store
Trans = State ——> State
ETrans = State ——> (State x Val)
OpDen = Val^n x Val^m
```

In the following we assume that the syntactic domain ID and the semantic domain Id are identical.

2.2.3.2. The memory model

In the semantic clauses a two-level memory model is used. The first level, represented by the domain Env of environments, links identifier to a vector of values. One of them is a location of a (virtual) memory, in which the associated value is stored. This represents the second level of the memory model, and it is formed by the domain Store.

Using $\xi \in$ Env, $\sigma \in$ Store we have:

$$\begin{tikzcd}
\text{id} \arrow[r, "\xi"'] & (\text{location}, \ldots, \ldots) \arrow[d, "\sigma"] \\
 & \langle\text{value}\rangle
\end{tikzcd}$$

For the different kinds of object qualifications the following memory schemes are used:

$\xi(\text{id})\downarrow 2 \in$ AlgQual:

$$\begin{tikzcd}
\text{id} \arrow[r] & (\text{Loc}, \text{obq}, V \in \text{ValQual}) \arrow[d] \\
 & A \in \text{Alg}
\end{tikzcd}$$

with TOI(A)↓1 = V

$\xi(id)\downarrow 2$ = VAR:

$$\begin{tikzcd}
id \arrow[r] & (loc,\ VAR,\ V \in ValQual) \arrow[d] \\
 & v \in V
\end{tikzcd}$$

with $TOI(A)\downarrow 1 = V$

$\xi(id)\downarrow 2 \in \{PROC,\ FUNC\}$:

$$\begin{tikzcd}
id \arrow[r] & (loc,\ PROC/FUNC,\ \perp) \arrow[d] \\
 & D \in OpDen
\end{tikzcd}$$

2.2.4. The Main Program Algebra

The semantic clauses of sec. 3. state the meaning of type definitions, procedure and function declarations, and so on. Each clause that describes the introduction of a new item includes an updating of a so-called main program algebra (MPA) that is accessable in every $\xi \in$ Env under the reserved standard identifier 'main'.

MPA serves as a vehicle to express the effect of a program by algebraic means. The induced state transformation is formulated as an algebra operation where the argument sets are determined by algebra carriers.

The construction of MPA is incremental: initially 'main' is bound to the algebra PRE $\in$ Alg (see sec. 2.2.2.) to model the set of predefined objects of ModPascal at the beginning of every computation. Skipping label and constant declarations of a program P at this point, the object definitions of (objectL P) - the type/enrichment/instantiation-part of [Olt 84] - are elaborated firstly. In the ModPascal semantics, each type or enrichment definition leads to an algebra A that is stored under the definition identifier. Simultaneously, the set of visible objects is enlarged by the current definition, and this fact is taken into account by uniting the main program algebra of the current state (ξ, σ) with the new algebra:

$$\xi(main)\downarrow 1 := \xi(main)\downarrow 1 \cup A$$

The instantiation object definitions are treated not in that way since they represent a kind of meta-objects: instead of mapping carrier elements to carrier elements they map carriers to carriers and operations to operations (see sec. 3.6.). Because this would extend the MPA concept without giving profit in some of the intended MPA applications (see below) we disregard the embedding of instantiation objects in MPA.

The procedure and function declarations of (subprogL P) are mapped to algebra functions F by the ModPascal semantics. The carriers of the source ($\equiv$ the parameter types of the declarations) have to be visible at the declaration point. But this is equivalent to require each source set to be contained in $\xi(\text{main})\downarrow 1$. Therefore the addition of F to the current main program algebra operations is well-defined, and this action is performed for each declaration in (subprogL P):

$$\xi(\text{main})\downarrow 1 := \xi(\text{main})\downarrow 1 \cup (\emptyset, \{F\})$$

In parallel, the meaning of an operation declaration is also stored under the operation identifier (see sec. 2.2.3.2.). This is true too for operations introduced in module type or enrichment definitions. The redundancy offers some conveniance in the formulation of semantic clauses, but has no theoretical benefit.

The statement list (stmtL P) is viewed at as a specific procedure body, where the variables of (varL P) represent the local variables and where - for simplicity - input/output behaviour is disregarded. Then the usual treatment of procedure declarations is performed, the resulting algebra operation S is stored under the reserved standard identifier 'stmtproc' and added to the main program algebra:

$$\xi(\text{main})\downarrow 1 := \xi(\text{main})\downarrow 1 \cup (\emptyset, S)$$

In some sence 'stmtproc' is the only operation of a ModPascal program. If every function or procedure declaration would be embedded in a module type or enrichment definition such that all parameter types are used by the definition - that is to associate every operation with an object -, then the main program would consist only of object definitions and the only operation 'stmtproc'. This view also includes a hierarchical structure lying on all program objects. In the following a special structure of 'main' is not assumed.

Before describing the benefits of the main program algebra concept it should be briefly mentioned that label and constant definitions fit into this framework. Labels are just special constants, and constants themselves can be modelled as no-argument functions yielding the constant value. Because constants are of a specific type, there is an algebra that can be enlarged by the associated no-argument function, and by this $\xi(\text{main})\downarrow 1$ is enlarged.

The main program algebra construction is very helpful in the verification of (concrete) ModPascal programs against (abstract) specifications (see [Olt 85]). Especially algebraic specifications are a concise and mathematically sound method of describing what a program should do, and the theory of abstract data types is based upon them. If verification tasks are performed in this setting, one profits from the following points:

- The semantics of the concrete program and the abstract specification are both algebras. Instead of checking verification conditions on program texts, algebraic structures and methods can be used easily since only one formal system is involved. Furthermore, universal algebra comes with a much broader set of possible correctness criteria (e.g. homomorphisms, isomorphisms, generating sets) than conventional Hoare-style logic (derivability).

- The object view on types has an analogon in algebraic specifications: each describes a specific set of data and operations. But without the main program algebra it would be impossible to provide an appropriate meaning to the statement part of a program, and therefore program and specification would become incomparable.
- Main program algebras allow to treat programs as objects. There is no difference between the semantics of a module and of a prog, such that the features as 'hierarchization of programs' or 'separate compilation' could be provided with a clear formal semantics.
- For applications in special verification contexts it is necessary to precompile ModPascal code to Pascal code (see sec. 4.). To get a conveniant notion of semantical preservation in this process, the main program algebra is used as an important idea. It helps to express conditions for the states resulting from the elaboration of the ModPascal as well as of the Pascal construct.

The exploitation of these facts would go far beyond the intention of this paper. In section 4. only the precompilation aspect is examined. For our purposes it is sufficient to have an informal idea of the main program algebra and its justification. In [Olt 85] this concept is applied.

2.2.5. Environment and Data Base

Before giving the semantic clauses for ModPascal we will shortly scetch some practical problems that arise if the currently available implementation of ModPascal is used. As described in the introduction ModPascal is back-end of a verifiable software development process supported by the ISDV. There is a ModPascal Programming System (MPPS) inside the ISDV consisting of an editor, a ModPascal-to-Pascal precompiler, a Pascal compiler and an execution device. All components refer to a data base (DB) in which ModPascal objects (modules, enrichments, instantiations, standard types, ...) are administrated. Thus, modelling the behaviour of ModPascal by using the two level memory model described above includes modelling the behaviour of DB in MPPS.

To illustrate this we give a short example. MPPS distinguishes different users each of them having a separate section of objects in the DB. If new objects are entered, there are two ways of involving other objects into the current one: either they are defined explicitly in the current input, so that they are directly visible (declaration-before-use paradigm), or they are referred to via the DB. In the last case the system checks if the access to the desired objects is allowed. The permission of using objects is qualified by different categories. Any user has unlimited access to his own objects, but can also use, read or read/write objects of other users, or use system objects (for details, see [RL 84]). So, if the access to the referenced object of DB is allowed, it is incorporated in the current computation.

The semantic domain Env is an abstraction of the DB. Since ξ(id) ∈ (Loc x ObQual x ValQual), there are no object specific access rights in the model, and in fact, the concept of multi-user and separate object spaces is disregarded. But this is not a serious disadvantage, because it could be easy taken into our model by adding appropriate domains and modifying the semantic clauses below. In the special case of semantic clauses for instantiate type definition, this model extension is actually performed (see sec. 3.7.).

The decision not to bother with data organization questions in the ModPascal semantics removes a degree of complexity and enables a more succint description of how things are intended to work.

2.3. Semantic Functions

2.3.1. Main Functions

The syntactic and semantic domains are linked by the following functions:

(a) M: Constr ---> State ---> State

where Constr = Program + Prog_head + Block + Lab + ... is the sum of all syntactic domains.

<u>Notation</u>: Elements of Constr will be enclosed in double brackets $[\![$ and $]\!]$. Elements (ξ, σ) of State will be supplied to M with juxtaposed components.
Example: $M[\![c]\!]\xi\sigma$

M links an initial state prior execution of a programming language construct to a final state after execution of this construct. M is defined by the semantic clauses of sec. 3. which are elaborated to an appropriate level of detail.

M is applicable to every c ∈ Constr except the cases listed below:

<u>C ∈ Expr</u>:

(b) E: Expr ---> State ---> (State ---> Val)
and $M[\![c]\!]\xi\sigma \Rightarrow E[\![c]\!]\xi\sigma$

<u>C ∈ (Stand_type v Stand_type_gen)</u>:

(c) Mt: (Stand_type v Stand_type_gen) ---> State ---> (ObQual x ValQual x Alg)
and $M[\![c]\!]\xi\sigma \Rightarrow Mt[\![c]\!]\xi\sigma$

c ∈ Module_type:

```
(d) Mm: Module_type ---> State
              ---> ((ObQual x ValQual x Alg) x State)
    and $M[\![c]\!]\xi\sigma \Rightarrow Mm[\![c]\!]\xi\sigma$
```

C ∈ Enrich_def:

```
(e) Me: Enrich_def ---> State ---> State
    and $M[\![c]\!]\xi\sigma \Rightarrow Me[\![c]\!]\xi\sigma$
```

C ∈ Instantiate_type:

```
(f) Mi:  Instantiate_type ---> State --->  ((ObQual x ValQual  x
    Alg) x State)
    and $M[\![c]\!]\xi\sigma \Rightarrow Mi[\![c]\!]\xi\sigma$
```

2.3.2. Auxiliary Functions

The following functions are used as auxiliary functions in sec. 3.

newLoc
newLoc gets a currently unused Location of an environment.

```
   newLoc: Env ---> Loc
   newLoc($\xi$) := $\eta$ Loc . $\forall$ id $\in$ Id . $\xi$(id)$\downarrow$1 $\neq$ Loc
```

searchdef
searchdef Looks for the algebra to which an operation is associated; it returns the algebra identifier.

```
   searchdef: Id ---> State ---> Id
   searchdef(opid)$\xi\sigma$ :=
      let id := $\iota$ $id_1$ $\in$ Id . $\xi(id_1)\downarrow 2$ $\in$ AlgQual and
                                  let (C, F) := $\sigma(\xi(id_1)\downarrow 1)$ in
                                     opid $\in$ opnames(F) in
           id
```

(ι returns $\perp_{Id}$ if no unique id_1 exists with the required property)

standard
indicates whether an identifier denotes a standard object, and provides its initialization value in the positive case.

```
   standard: Id ---> (D_Bool x Val)
   standard(id) :=
              if id = BOOL ---> (true, false) else
              if id = INT ---> (true, 0) else
                  .
                  .
                  .
              else (false, $\perp$)
```

3. Semantic Clauses

The semantic clauses of this section are stated in a way that avoids too much complexity induced by a treatment of all important aspects of a language description. There are the following restrictions and modifications:

- Not every syntactical construct of 2.1.2. is supplied with a semantical clause. Very often there are only minor changes in the ModPascal semantics for Standard Pascal constructs, such that descriptions as [Ten 76] may suffice for an understanding. Also, it is not the topic of this paper to describe for example the 'repeat' construct.
  We will concentrate the definition of clauses on those constructs that make ModPascal differ from Pascal and those Pascal constructs with major deviations from their usual semantics.
- Nothing will said about scoping, type checking or coercions.
- Since we omit jumps and expression error handling, the need of continuations and expression continuations does not arise. We refer to [Ten 76] or [Sto 77] for an appropriate treatment of the respective ModPascal constructs.
- The dynamic behaviour induced by environment changes is not modelled here. One effect of such an action is a changed semantics of identifiers because another scope is enforced by the new environment. This should take over to the algebraic semantics and it would mean to install a new program algebra with possibly new interpretations of its function symbols. More general, environment changes correspond to algebra changes, and beside the boring technical issues we do not want to develop a theory of it here.

3.1. Procedures and Functions

3.1.1. Declarations

In ModPascal, procedures as well as functions may have side-effects on the embedding environment. The side-effect can be formalized in the state change of global variables, under respection of domain structure of value sets of variables. The semantics of procedure and function declarations then is based upon the side-effect formalization associated to the operation body.

The provision of a clear formalism is the first task of this section.
Types will be associated with algebras, and variables of a type will take values in a specific carrier set associated to the algebra of the type (the TOI). Then, if the effect of an operation call is described in the state change of its global variables, this can be modelled as an assignment of new TOI values to the variables. More precise: Let op denote an operator with global variables gl_1, ..., gl_n of types T_1, ..., T_n. Let A_i denote the semantical algebra behind T_i, and V_i := TOI(A_i), i ∈ (n). Then the values of gl_i in a given state (ξ, σ) (i.e. $\sigma(\xi(gl_i)\downarrow 1)$) are elements of V_i, i ∈ (n). Under the

assumption that the global variable set of op remains constant during operation execution, the pairs (vec_1, vec_2) describe the operations behaviour, where vec_1 denotes the vector of global variable values <u>before</u> execution, and vec_2 <u>after</u>. Since vec_i, $i \in \{1, 2\}$ are n-tuples of $V_1 \times \ldots \times V_n$, they describe the semantic relation associated to op, and in the case of deterministic languages, this relation is a function.

With this in mind there is a technique to link the ModPascal operation declaration to a semantic function of an appropriate order algebra (that at least contains V_i, $i \in (n)$ among its carrier sets):

1) Generate pairs (vec_1, vec_2) that are intended to describe the operations behaviour. Let Semop denote the set of all generated pairs.
2) If Semop denotes not a strict function: make it strict by exchanging strictness-violating pairs through strictness-preserving ones.
3) Take an algebra in which Semop is defined, i.e. for $i \in (n)$ V_i is a carrier set.

ad1) In our setting, the pair-generating mechanism is the semantic function M. We consider its values for a source and target state, and from there we derive values of global variables of the construct that caused the state change. This process is done for <u>all</u> possible values of global variables in the source state. In other words, this means that the elaboration of M on operation declarations is reformulated in terms of algebra functions. This reformulation involves a recursive process since ModPascal operation declarations may be recursive such that fixpoints have to be taken as the solution of the reformulation task (see e.g. Sem_1, Sem_2, Sem_15 and Sem_16 below).
Pictorially, we have

$$\begin{tikzcd}
(vec_1) \arrow[r, dashed] \arrow[d] & (vec_2) \\
(\xi, \sigma) \arrow[r, "M"] & (\xi_1, \sigma_1) \arrow[u]
\end{tikzcd}$$

where - - -> denotes Semop.

ad2) To achieve strictness, two ways are possible: the faulty construct is redefined by the programmer, or the semantics is automatically manipulated in order to regain strictness. Even if the second choice is unpleasant for its unvisible redefining, the processes defined below take it.

ad3) The semantic algebra will not be constructed for a single operation declaration. Instead of, MPA will serve as structure for the embedding of Semop. Since ModPascal follows the 'declaration-before-use' paradigm, the definedness requirement for V_i will be satisfied for syntactic correct programs.

It should be emphasized that this view of the semantics of operations (as algebra functions) has not been much developed in the denotational semantics literature. One of the reasons probably was the lack of a satisfactory concept of types in semantic domains (e.g. [Don 77], [Rey 74]) until abstract data type theory come up in the second half of the 1970s. The model developed here tries to union both concepts.

The following definitions will in the first instance distinguish between procedures and functions. In the case of procedures only state changes are considered, in the case of functions state changes and the computation of a result value (despite the fact that later function calls will have to obey the side-effect-freeness condition 3.2.1.-2, here the general approach is taken, that is also more close to the (Mod)Pascal reality). The definitions of the technical operators $\mathcal{R}_P$, $\mathcal{R}_F$, $\mathcal{R}_P{}^*$, $\mathcal{R}_F{}^*$, ${}_P[\_]$, ${}_F[\_]$, ${}_{PF}[\_]$, $\mathcal{R}$ are intended to achieve steps 1) and 2) above in the ModPascal environment. Their differences lie in the syntactic domains they are defined upon:

$\mathcal{R}_P$	:	Procedure declarations
$\mathcal{R}_F$	:	Function declarations
$\mathcal{R}_P{}^*$	:	Strictness generating version of $\mathcal{R}_P$
$\mathcal{R}_F{}^*$	:	Strictness generating version of $\mathcal{R}_F$
${}_P[\_]$	:	Sets of (mutually recursive) procedure declarations
${}_F[\_]$	:	Sets of (mutually recursive) function declarations
${}_{PF}[\_]$	:	Sets of (mutually recursive) procedure and function declarations
$\mathcal{R}$	:	A switch operator which branches for arbitrary sets of operation declarations to the appropriate operator

These operators are applied in the semantic clauses Sem_1, Sem_2, Sem_15 and Sem_16 below.

<u>Def. 3.1.1.-1</u>. [$\mathcal{R}_P$, $\mathcal{R}_F$]
Let $Gl_1 := \{id_1, \ldots, id_n\} \subseteq Id$, $Gl_2 := \{id_1', \ldots, id_m'\} \subseteq Id$.
For $id \in (Gl_1 \cup Gl_2)$ and $\xi \in Env$, a value set V_{Id} is associated to id if

1) $\xi(id)\downarrow 2 = VAR$
2) $\xi(id)\downarrow 3 = V_{Id}$

Let $V(Gl_1, Gl_2) := V_{Id_1} \times \ldots \times V_{Id_n} \times V_{Id'_1} \times \ldots \times V_{Id'_m}$,
where $\xi(id_i) \neq \perp$, $\xi(id_j') \neq \perp$, $i \in (n)$, $j \in (m)$.
Let $\bar{V} := \{V(Gl_1, Gl_2) \mid Gl_1 \subseteq Id, Gl_2 \subseteq Id\}$ (Set of n-ary cartesian products). Let V_r denote a value set.

1) $\mathcal{R}_P$: $Trans \times Env \times 2^{Id} \times 2^{Id} \longrightarrow \bar{V}$ with
$\mathcal{R}_P(T, \xi, Gl_1, Gl_2)$:= the least relation on $V(Gl_1, Gl_2)$ defined by

```
(1) $\forall\, id \in (Gl_1 \cup Gl_2) \,.\, \xi(id)\downarrow 3 = V_{Id}$
(2) $\forall\, \sigma \in Store \,.$
        Let $(\bar{\xi}, \bar{\sigma}) := T(\xi, \sigma)$ in
        Let $x_i := \sigma(\xi(id_i)\downarrow 1)$, $i \in (n)$ in
        Let $y_i := \bar{\sigma}(\bar{\xi}(id'_i)\downarrow 1)$, $i \in (m)$ in
            $(x_1, \ldots, x_n, y_1, \ldots, y_m) \in \mathcal{R}_P(T, \xi, Gl_1, Gl_2)$
```

is called <u>store transformation</u> (w.r.t. Gl_1, Gl_2 and T).

2) $\mathcal{R}_F$: ETrans x Env x 2^{Id} x 2^{Id} x Val ⟶ $\bar{V}$ x Val
$\mathcal{R}_F(E, \xi, GL_1, GL_2, V_r)$:= the least relation on $(V(GL_1, GL_2) \times V_r)$ defined by
(1) $\forall$ id $\in (GL_1 \cup GL_2)$. $\xi(id) \neq \bot Env$
(2) $\forall \sigma \in$ Store .
let $r \in V_r$ in
let $((\bar{\xi}, \bar{\sigma}), r) := E(\xi, \sigma)$ in
let $x_i := \sigma(\xi(id_i)\downarrow 1)$, $i \in (n)$ in
let $y_i := \bar{\xi}(\bar{\sigma}(id'_i)\downarrow 1)$, $i \in (m)$ in
$(x_1, ..., x_n, y_1, ..., y_m, r) \in \mathcal{R}_F(E, \xi, GL_1, GL_2, V_r)$
is called <u>store transformation with result</u> (w.r.t. GL_1, GL_2, E). ¤

Notation: If no ambiguities are possible $\mathcal{R}_P$ denotes the store transformation and $\mathcal{R}_F$ the store transformation with result.

<u>Fact 3.1.1.-2.</u>: $\mathcal{R}_P$, $\mathcal{R}_F$ are functions. ¤

The quantities of the definition could be interpreted as follows:

- GL_1, GL_2 : set of program variables on which the effects of the execution of the operation body are investigated
- V_{id} : value set for program variable id
- V_r : value set for a function result
- T ∈ Trans : the state transformation induced by the procedure body
- E ∈ ETrans : the state transformation induced by the function body and an evaluated result

In some sense, $\mathcal{R}_P / \mathcal{R}_F$ are restrictions of T/E, since for $GL_1 = GL_2 = Id$, $\mathcal{R}_P$ and T as well as $\mathcal{R}_F$ and E are identical for fixed environments ξ if only variable values are regarded.

From $\mathcal{R}_P / \mathcal{R}_F$ it is easy to generate a relation on carriers of algebras. For example, for $\mathcal{R}_P$ it holds that
$\mathcal{R}_P(T, \xi, GL_1, GL_2) \subseteq V_{id_1} \times ... \times V_{id_n} \times V_{id'_1} \times ... V_{id'_m}$
and each V_{id} directly corresponds to some algebra $A \in Alg[\Sigma]$ with $TOI(A)\downarrow 1 = V_{id}$. In other words, if $\mathcal{R}_P / \mathcal{R}_F$ turn out to be <u>strict</u> functions, an order algebra A = (C, F) may be constructed with $\mathcal{R}_P \in F$ / $\mathcal{R}_F \in F$.

<u>Def. 3.1.1.-3.</u> $[\mathcal{R}_P{}^*, \mathcal{R}_F{}^*]$
Let $\mathcal{R}_P(T, \xi, GL_1, GL_2)$ and $\mathcal{R}_F(E, \xi, GL_1, GL_2, V_r)$ be defined as in definition 3.1.1.-1.
1) $\mathcal{R}_P{}^*(T, \xi, GL_1, GL_2)$ is defined by:
$\forall (x_1, ..., x_n, y_1, ..., y_m) \in \mathcal{R}_P(T, \xi, GL_1, GL_2)$.
1) $\forall i \in (n)$. $x_i \neq \bot V_{id_i}$
$\Rightarrow (x_1, ..., x_n, y_1, ..., y_m) \in \mathcal{R}_P{}^*(T, \xi, GL_1, GL_2)$
2) $\exists i \in (n)$. $x_i = \bot V_{id_i}$
$\Rightarrow (x_1, ..., x_n, \bot V_{id'_1}, ..., \bot V_{id'_n}) \in \mathcal{R}_P{}^*(T, \xi, GL_1, GL_2)$.

2) $\mathcal{R}_F{}^*(E, \varrho, Gl_1, Gl_2, V_r)$ is defined by:
$\forall (x_1, \ldots, x_n, y_1, \ldots, y_m, r) \in \mathcal{R}_F(E, \varrho, Gl_1, Gl_2, V_r).$
1) $\forall\ i \in (n)\ .\ x_i \neq \bot V_{Id\,i}$
$\Rightarrow (x_1, \ldots, x_n, y_1, \ldots, y_m, r) \in \mathcal{R}_F{}^*(E, \varrho, Gl_1, Gl_2, V_r)$
2) $\exists\ i \in (n)\ .\ x_i = \bot V_{Id\,i}$
$\Rightarrow (x_1, \ldots, x_n, \bot V_{Id'\,i}, \ldots, \bot V_{Id'\,m}, \bot Vr) \in \mathcal{R}_F{}^*(E, \varrho, Gl_1, Gl_2, V_r).$ ¤

Definition 3.1.1.-3. shows how to augment $\mathcal{R}_P / \mathcal{R}_F$ to make them strict. But frequently $\mathcal{R}_P = \mathcal{R}_P{}^* \ /\ \mathcal{R}_F = \mathcal{R}_F{}^*$, i.e. the store transformations are already strict.

A very important extension of $\mathcal{R}_P{}^*/\mathcal{R}_F{}^*$ is the case of sets of state transitions. This models the situation that the state change is caused by more than one function, and it covers also mutual dependencies between the elements of the state transition set.

Def. 3.1.1.-4 [extended state transition]

1) Let $\{T_1, \ldots, T_n\} \subseteq$ Trans, $n \in \mathbb{N}$. Then
$_P[\_]: \mathcal{P}(\text{Trans}) \longrightarrow \text{Trans}$
denotes the extended P-state transition defined by
$_P[\{T_1, \ldots, T_n\}]\varrho\sigma :=$

```
┌ $(\varrho', \sigma')$ if $\exists\ k \in \mathbb{N}, (\varrho_i, \sigma_i) \in$ State, $i \in (k)$ .
│                 let $(\varrho_0, \sigma_0) := (\varrho, \sigma)$,
│                     $(\varrho_K, \sigma_K) := (\varrho', \sigma')$        in
┤                   $\forall\ i \in (k-1)\ .\ \hat{\exists}\ j \in (n)$ .
│                      $(\varrho_{i+1}, \sigma_{i+1}) = T_j(\varrho_i, \sigma_i)$
│
└ $\bot$          otherwise
```

2) Let $\{E_1, \ldots, E_n\} \subseteq$ ETrans, $n \in \mathbb{N}$. Then
$_F[\_]: \mathcal{P}(\text{ETrans}) \longrightarrow \text{ETrans}$
denotes the extended F-state transition defined by
$_F[\{E_1, \ldots, E_n\}]\varrho\sigma :=$

```
┌ $((\varrho', \sigma'), e')$ if $\exists\ k \in \mathbb{N}, (\varrho_i, \sigma_i) \in$ State,
│                                         $e_i \in$ Val, $i \in (k)$ .
│                     let $(\varrho_0, \sigma_0) := (\varrho, \sigma)$, $(\varrho_K, \sigma_K) :=$
┤                                $(\varrho', \sigma')$, $e_K := e'$ in
│                       $\forall\ i \in (k-1)\ .\ \hat{\exists}\ j \in (n)$ .
│                          $(\varrho_{i+1}, \sigma_{i+1}) = E_j(\varrho_i, \sigma_i){\downarrow}1$ and
│                           $e' = E_j(\varrho_{K-1}, \sigma_{K-1}){\downarrow}2$
└ $\bot$                 otherwise
```

3) Let $\{ET_1, \ldots, ET_n\} \subseteq$ (Trans + ETrans), $n \in \mathbb{N}$. Then
$_{PF}[\_]: \mathcal{P}(\text{Trans} + \text{ETrans}) \longrightarrow (\text{Trans} + \text{ETrans})$
denotes the extended PF-state transition defined by
$_{PF}[\{ET_1, \ldots, ET_n\}]\varrho\sigma :=$

```
┌ $(\varrho', \sigma')$ if $\exists\ k \in \mathbb{N}, (\varrho_i, \sigma_i) \in$ State, $i \in (k)$ .
│            let $(\varrho_0, \sigma_0) := (\varrho, \sigma)$, $(\varrho_K, \sigma_K) := (\varrho', \sigma')$ in
│             $\forall\ i \in (k-1)\ .\ \hat{\exists}\ j \in (n)$ .
│             case $ET_j \in$ Trans: $(\varrho_{i+1}, \sigma_{i+1}) = ET_j(\varrho_i, \sigma_i)$
│             case $ET_j \in$ ETrans: $(\varrho_{i+1}, \sigma_{i+1}) = ET_j(\varrho_i, \sigma_i){\downarrow}1$
│            and $ET_{K-1} \in$ Trans
```

```
$((\xi', \sigma'), e')$ if $\exists\ k \in \mathbb{N}, (\xi_i, \sigma_i) \in State, e_i \in Val,$
                                              $i \in (k)$ .
          Let $(\xi_0, \sigma_0) := (\xi, \sigma), (\xi_K, \sigma_K) := (\xi', \sigma'),$
                                        $e_K := e'$ in
           $\forall\ i \in (k-1)\ .\ \hat{\exists}\ j \in (n)$ .
           case $ET_j \in Trans: (\xi_{i+1}, \sigma_{i+1}) = ET_j(\xi_i, \sigma_i)$
           case $ET_j \in ETrans: (\xi_{i+1}, \sigma_{i+1}) = ET_j(\xi_i, \sigma_i)\downarrow 1$
          and $ET_{K-1} \in ETrans$
 $\perp$       otherwise                                                      ¤
```

Notation: The curled brackets in $_P[...]$ and $_F[...]$ are omitted.

$_P[T_1, ..., T_n] / _F[E_1, ..., E_n]$ describe a composition of a finite sequence of state transitions, if defined. Each intermediate state is application argument to exactly one state transition. The appropriate selection can be thought as determined by the predecessing state. In the case of $[E_1, ..., E_n]$ the intermediate states can be thought as providing locations to store evaluated results if necessary.

With this definition $\mathcal{R}_P{}^*({}_P[T_1, ..., T_n], \xi, Gl_1, Gl_2)$ or $\mathcal{R}_F{}^*({}_F[E_1, ..., E_n], \xi, Gl_1, Gl_2, V_r)$ evaluate to the associated store transformation (with result).

These technical operations are now combined into a single function.

Def. 3.1.1.-5 [extended store transition]
Let $\bar{V}$ be defined as in 3.1.1.-1
Let TR := $\mathcal{P}(ETrans + Trans)$
    IV := $(2^{Id} \times 2^{Id}) + (2^{Id} \times 2^{Id} \times Val)$
    RES := $(\bar{V} + (\bar{V} \times Val))$
Then the extended store transformation $\mathcal{R}$: TR x Env x IV ---> RES is defined by

```
$\mathcal{R}(tr, \xi, iv) :=$
  case $iv = (Gl_1, Gl_2) \in (2^{Id} \times 2^{Id})$:
     case $tr = \{T\} \in Trans: \mathcal{R}_F{}^*(T, \xi, Gl_1, Gl_2)$
     case $tr = \{T_1, ..., T_n\} \subseteq Trans$:
                              $\mathcal{R}_P{}^*({}_P[T_1, ..., T_n], \xi, Gl_1, Gl_2)$
     case $tr = \{ET_1, ..., ET_n\}, {}_{PF}[ET_1, ..., ET_n] \in Trans$:
                   $\mathcal{R}_P{}^*({}_{PF}[ET_1, ..., ET_n], \xi, Gl_1, Gl_2)$
     otherwise $\perp$
  case $iv = (Gl_1, Gl_2, V_r) \in (2^{Id} \times 2^{Id} \times Val)$:
     case $tr = \{E\} \in ETrans: \mathcal{R}_F{}^*(E, \xi, Gl_1, Gl_2, V_r)$
     case $tr = \{E_1, ..., E_n\} \subseteq ETrans$:
                            $\mathcal{R}_F{}^*({}_F[E_1, ..., E_n], \xi, Gl_1, Gl_2, V_r)$
     case $tr = \{ET_1, ..., ET_n\}, {}_{PF}[ET_1, ..., ET_n] \in ETrans$:
                   $\mathcal{R}_F{}^*({}_{PF}[ET_1, ..., ET_n], \xi, Gl_1, Gl_2, V_r)$
     otherwise $\perp$
  otherwise $\perp$
                                                                           ¤
```

We are now ready to state the semantics of procedure declarations and function declarations. Both are, dependant of global variables and formal parameters, that are visible in the operation body, and results are delivered via global variables. In the case of functions, also a value is computed. We distinguish the following sets:

GL(op) : set of all (direct and indirect) global variables of op.
FP(op) : set of all formal parameters of op.
D(op) : GL(op) $\cup$ FP(op)

These sets will be supplied to the store transformations $\mathcal{R}_p{}^*$, $\mathcal{R}_f{}^*$. In a given state, all variables are associated to a fixed algebra and therefore to a fixed value set. The store transformations then generate functions from the cartesian product of D(op) to the cartesian product of GL(op).

Sem_1: Procedure declaration

```
M⟦p: Proc_dcl⟧$\varrho\sigma$ :=
                Let id := (proc_id p), bdy := (body p),
                    loc := newloc($\varrho$) in
                Let ST := fix T . $\lambda\varrho_1\sigma_1$ .
                    M⟦bdy⟧ $\varrho_1$ [id ↤ (loc, PROC,
                              $\mathcal{R}_p$*(T, $\varrho_1$, D(p), GL(p)))]$\sigma_1$
                    where ($\varrho_1$, $\sigma_1$) contains parameters
                    $\in$ (paraml p) after calling and
                    passing to the body in
                Let R := $\mathcal{R}_p$*(ST, $\varrho$, D(i), GL(i)) in
                Let $\varrho_2$ := $\varrho$[id ↤ (loc, PROC, $\emptyset$)],
                    $\sigma_2$ := $\sigma$[loc ↤ R,
                          $\varrho$(alg)↓1 ↤ $\sigma$($\varrho$(alg)↓1)
                                                   $\cup$ ($\emptyset$, R)]) in
                ($\varrho_2$, $\sigma_2$)
```

The store transformation of the procedure body is computed and assigned to the procedure identifier. Also the main program algebra is updated.

Sem_2: Function declaration

```
M⟦f: Func_dcl⟧$\varrho\sigma$ :=
                Let id := (func_id f), bdy := (body f),
                    loc := newloc($\varrho$) in
                Let $V_r$ := $\varrho$(result f)↓3 in
                Let ST := fix E . $\lambda\varrho_1\sigma_1$ .
                    M⟦bdy⟧ $\varrho_1$ [id ↤ (loc, FUNC,
                           $\mathcal{R}_f$*(E, $\varrho_1$, D(f), GL(f), $V_r$))]$\sigma_1$
                    where ($\varrho_1$, $\sigma_1$) contains parameters $\in$
                    (paraml f) after calling and
                    passing to the body in
                Let R := $\mathcal{R}_f$*(ST, $\varrho$, V(f), GL(f), $V_r$) in
                Let $\varrho_2$ := $\varrho$[id ↤ (loc, FUNC, $\emptyset$)],
                    $\sigma_2$ := $\sigma$[loc ↤ R,
                              $\xi(alg)\downarrow 1 \leftarrow \sigma(\xi(alg)\downarrow 1)$
                                                   $\cup\ (\phi, R)])$ in
             $(\xi_2, \sigma_2)$
```

The result set of the function is computed and passed to the store transformation generation. The outcome is assigned to the function identifier, and the main program algebra is updated.

3.1.2. Calls

To describe the meaning of a procedure or a function invocation the store transformations $\mathcal{R}_P{}^*/\mathcal{R}_F{}^*$ are applied. This reflects the fact that the meaning of a procedure or function call can be expressed in value changes of some global variables.

Def. 3.1.2.-1. [application store transformation]
Let $\mathcal{R}_P{}^*(T, \xi, Gl_1, Gl_2)$ and $\mathcal{R}_F{}^*(E, \xi, Gl_1, Gl_2, V_r)$ be defined as in 3.1.1.-1./3.
Then the application of the store transformation to states, i.e.

```
  $(2^{Id} \times 2^{Id}) \longrightarrow State \longrightarrow State$                  $(\mathcal{R}_P{}^*)$
  $(2^{Id} \times 2^{Id} \times Val) \longrightarrow State \longrightarrow (State \times Val)$     $(\mathcal{R}_F{}^*)$
```

is defined by:

```
1) $\mathcal{R}_P{}^*(T, \xi, Gl_1, Gl_2)\ (\xi, \sigma) :=$
        let $a := \iota\ V \in \mathcal{R}_P{}^*(T, \xi, Gl_1, Gl_2)\ .\ (\forall\ i \in (n)\ .$
                 $\sigma(\xi(id_i)\downarrow 1) = V\downarrow i)$ in
        let $\sigma' := \sigma[\xi(id_j')\downarrow 1 \longleftarrow V\downarrow(n+j)],\ j \in (m)$ in
            $(\xi, \sigma')$
2) $\mathcal{R}_F{}^*(E, \xi, Gl_1, Gl_2, V_r)(\xi, \sigma) :=$
        let $a := \iota\ V \in \mathcal{R}_F{}^*(E, \xi, Gl_1, Gl_2, V_r)\ .\ (\forall\ i \in (n)\ .$
                 $\sigma(\xi(id_i)\downarrow 1) = V\downarrow 1)$ in
        let $\sigma' := \sigma[\xi(id_j')\downarrow 1 \longleftarrow V\downarrow(n+j)],\ j \in (m)$ in
        let $r := V\downarrow(n+m+1)$ in
        $((\xi, \sigma'), r)$                                                ¤
```

Definition 3.1.2.-1. states the dynamic behaviour of store transformations. At first the appropriate element of $\mathcal{R}^*$ is chosen by looking at the argument positions, and then the result is installed in the store component of the state as a copy of the result positions of the choosen elements.

Analogously the applications are defined, if sets of state transitions are supplied to $\mathcal{R}_F{}^*/\mathcal{R}_P{}^*$. The cases of extended P-state transition $(\mathcal{R}_P{}^*({}_P[T_1, \ldots, T_n], \xi, Gl_1, Gl_2)\xi\sigma)$, extended F-state transition $\mathcal{R}_F{}^*({}_F[E_1, \ldots, E_n], \xi, Gl_1, Gl_2, V_r)\xi\sigma)$ and extended PF-state transition $(\mathcal{R}({}_{PF}[ET_1, \ldots, ET_n]))$ can also be treated as a selection of appropriate vector components and their installation in a specific state, and this technical actions are performed by appropriate switching to $\mathcal{R}_F{}^*/\mathcal{R}_P{}^*$ inside $\mathcal{R}$:

Def. 3.1.2.-2 [application $\mathcal{R}$]
Let $\mathcal{R}$ be defined as in definition 3.1.1.-5.
Then the application of the extended store transformation $\mathcal{R}$ to states is defined by:

```
$\mathcal{R}$: (TR x Env x Id) ⟶ State ⟶ (State + (State x Val)) with
$\mathcal{R}(tr, \xi, iv)\xi\sigma$ :=
  case $iv = (Gl_1, Gl_2) \in (2^{Id} \times 2^{Id})$:
     case $tr = \{T\} \in$ Trans: $\mathcal{R}_F{}^*(T, \xi, Gl_1, Gl_2)\xi\sigma$
     case $tr = \{T_1, ..., T_n\} \subseteq$ Trans:
                                 $\mathcal{R}_P{}^*({}_P[T_1, ..., T_n], \xi, Gl_1, Gl_2)\xi\sigma$
     case $tr = \{ET_1, ..., ET_n\}, {}_{PF}[ET_1, ..., ET_n] \in$ Trans:
                      $\mathcal{R}_P{}^*({}_{PF}[ET_1, ..., ET_n], \xi, Gl_1, Gl_2)\xi\sigma$
     otherwise $\perp$
  case $iv = (Gl_1, Gl_2, V_r) \in (2^{Id} \times 2^{Id} \times Val)$:
     case $tr = \{E\} \in$ ETrans: $\mathcal{R}_F{}^*(E, \xi, Gl_1, Gl_2, V_r)\xi\sigma$
     case $tr = \{E_1, ..., E_n\} \subseteq$ ETrans:
                      $\mathcal{R}_F{}^*({}_F[E_1, ..., E_n], \xi, Gl_1, Gl_2, V_r)\xi\sigma$
     case $tr = \{ET_1, ..., ET_n\}, {}_{PF}[ET_1, ..., ET_n] \in$ ETrans:
                      $\mathcal{R}_F{}^*({}_{PF}[ET_1, ..., ET_n], \xi, Gl_1, Gl_2, V_r)\xi\sigma$
     otherwise $\perp$
  otherwise $\perp$
```

¤

This makes the meaning of operation declarations usable when the meaning of operation calls are computed.

In this semantic description of ModPascal we do not state semantic clauses that treat parameter evaluation and passing mechanisms (see the introduction for reasons of this confinement). Instead of we make assumptions that allow a convenient description of those effects that are of interest in our context. Speaking roughly, the assumptions concern the elaboration of expressions and consist mainly of the non-occurrence of side-effects in expression evaluations. For the non-Pascal partion of ModPascal this is quite realistic because functions of modules are defined in a way that allows easily to capsulate the occurring side effects on global variables (see sec. 4 and [Olt 84]). But Standard Pascal functions make more trouble in general since no restrictions are imposed on them concerning side effects (sometimes Pascal functions are called "procedures with value"). In the following semantic clauses we abstract from side effects and assume expressions of actual parameter lists of operation calls of benign character.

Assumption 3.1.2.-2.:
Let s:S denote a structure with (act_paramL s) defined.
Let $(e_1, ..., e_n)$:= (act_paramL s).
Let $((\xi_i, \sigma_i), r_i) := \mathcal{E}[\![e_i]\!]\xi\sigma$, $i \in (n)$ for given $(\xi, \sigma) \in$ State.
Then we assume $\xi_i = \xi_j$ and $\sigma_i = \sigma_j$ for $i, j \in (n)$. ¤

Sem_3: Procedure call

```
Proc_stmt = Term
Term = Simple_term ∨ Op_designator

M⟦s: Simple_term⟧$\varrho\sigma$ :=
 Let id :=(op_id s), ($e_1$, ..., $e_n$) := (act_paramL s) in
  Let ob := searchdef(id)$\varrho\sigma$ in
   if not (ob ≠ ⊥ENV and $\varrho$(id)↓2 = PROG) then error else
   else Let ($\varrho_1$, $\sigma_1$) := the state after elaboration of
                        calling and passing mechanisms for
                        ($e_1$, ..., $e_n$) in
    Let R := $\sigma_1$($\varrho_1$(id)↓1) in
     Let ($\varrho_2$, $\sigma_2$) := R($\varrho_1$, $\sigma_1$) in
      ($\varrho_2$, $\sigma_2$)

M⟦op: Op_designator⟧$\varrho\sigma$ :=
  Let $V_{id}$ := (var_id op), opid := (first (op_idL op)),
      ($e_1$, ..., $e_n$) := (first (act_paramL op))              in
  Let (loc, obq, vq) := $\varrho$($V_{id}$)
     if not (obq ∈ AlgQual) then error else
       Let (C, F) := $\sigma$($\varrho$($V_{id}$)↓1) in
        if not (opid ∈ opnames(F)) then error else
         Let ($\varrho_1$, $\sigma_1$) := the state after elaboration of
                            calling and passing mechanisms for
                            ($e_1$, ..., $e_n$)                in
             case $\varrho$(opid)↓2 = PROC :
                   Let R := $\sigma_1$($\varrho_1$(opid)↓1) in
                   Let ($\varrho_2$, $\sigma_2$) := R($\varrho_1$, $\sigma_1$), where
                       references to $id_i$ ∈ Gl(opid) are
                       substituted by references to
                       $\sigma_1$($\varrho_1$($V_{id}$)↓1)↓i              in
                   if (rest(op_idL op) = ∅ then
                       ($\varrho_2$, $\sigma_2$)
                   else Let op' := μ(op;
                                 op_idL:(rest(op_idL op)),
                          act_paramL:(rest(act_paramL op)))
                                                               in
                             M⟦op'⟧$\varrho_2\sigma_2$
             case $\varrho$(opid)↓2 = FUNC :
                Let R := $\sigma_1$($\varrho_1$(opid)↓1) in
                 Let (($\varrho_2$, $\sigma_2$), r) := R($\varrho_1$, $\sigma_1$), where
                            references to $id_i$ ∈ Gl(opid)
                            are substituted by references
                            to $\sigma_1$($\varrho_1$($V_{id}$)↓1)↓i       in
                  if (rest(op_idL op)) = ∅ then error
                  else
                  Let op' := μ(op; var_id: r,
                                 op_idL: (rest(op_idL op)),
                            act_paramL: (rest(act_paramL op)))
                  in
                              M⟦op'⟧$\varrho_2\sigma_2$
```

In the case of simple terms (= no dot notation occurs) the store transformation is applied directly. If opdesignators constitute a procedure call the sequence is elaborated step-by-step, and the result is only non-erroneous if the sequence ends with procedure invocation. Intermediate structures are composed via the µ-operator (see sec. 2.1.).

The correctness checks for operation call sequences (i.e. obq ∈ AlgQual and opid ∈ opnames(F)) have to be seen in connection with the static semantics for sequences as given in [OLt 84].

Sem_4: Function call

```
(Function call ≡ expression)
Expr = Id ∨ Term ∨ S_Term
Term = Simple_Term ∨ Op_designator

$E[\![id: Id]\!]\xi\sigma$ := if $\xi(id)\downarrow 2$ = VAR then $\sigma(\xi(id)\downarrow 1)$ else error

$E[\![t: Simple\_Term]\!]\xi\sigma$ :=
  let opid := (op_id t), $(e_1, ..., e_n)$ := (act_paramL t) in
   let $(\xi_1, \sigma_1)$ := the state after $E[\![e_i]\!]\xi\sigma$ and passing
                     the result to the function body  in
     if not $(\xi(opid)\downarrow 2$ = FUNC) then error else
          let R := $\sigma_1(\xi_1(opid)\downarrow 1)$   in
            let $((\xi_2, \sigma_2), r) := R(\xi_1, \sigma_1)$   in
                $((\xi_2, \sigma_2), r)$

$E[\![op: Op\_designator]\!]\xi\sigma$ :=
  let $V_{id}$ := (var_id op), opid := (first (op_idL op)),
        $(e_1, ..., e_n)$ := (first (act_paramL op))          in
  let (loc, obq, vq) := $\xi(V_{id})$
     if not (obq ∈ AlgQual) then error else
       let (C, F) := $\sigma(\xi(V_{id})\downarrow 1)$ in
        if not (opid ∈ opnames(F)) then error else
         let $(\xi_1, \sigma_1)$ := the state after elaboration of
                             calling and passing mechanisms for
                             $(e_1, ..., e_n)$                  in
             case $\xi(opid)\downarrow 2$ = PROC :
                         let R := $\sigma_1(\xi_1(opid)\downarrow 1)$ in
                         let $(\xi_2, \sigma_2) := R(\xi_1, \sigma_1)$, where
                             references to $id_i$ ∈ GL(opid) are
                             substituted by references to
                             $\sigma_1(\xi_1(V_{id})\downarrow 1)\downarrow i$           in
                             if (rest(op_idL op)) = ∅ then
                                error
                             else let op' :=
                          µ(op; op_idL:(rest(op_idL op)),
                             act_paramL:(rest(act_paramL op)))
                             in
                                 $E[\![op']\!]\xi_2\sigma_2$
             case $\xi(opid)\downarrow 2$ = FUNC :
                 let R := $\sigma_1(\xi_1(opid)\downarrow 1)$ in
                  let $((\xi_2, \sigma_2), r) := R(\xi_1, \sigma_1)$, where
                             references to $id_i$ ∈ GL(opid)
                             are substituted by references
```

```
                            to $\sigma_1(\varrho_1(V_{id})\downarrow 1)\downarrow i$          in
                      if (rest(op_idL op) = ∅ then
                          $((\varrho_2, \sigma_2), r)$
                      eLse
                      Let op' := µ(op; var_id: r,
                                   op_idL: (rest(op_idL op)),
                        act_paramL: (rest(act_paramL op)))
                      in
                                $E[\![op']\!]\varrho_2\sigma_2$

E⟦s: S_Term⟧ϱσ :=
      Let signum := (signum s), t := (term s) in
       Let $((\varrho_1, \sigma_1), r)$ := E⟦t⟧ϱσ in
        Let $r_1$ := "signum r" in
             $((\varrho_1, \sigma_1), r_1)$
```

The signum operator assigns a '+' or '-' to a term.

The function caLL cLause is equivaLent to the expression cLause. Again, the store transformation is appLied directLy if simpLe terms occur. Otherwise a sequence of operation caLLs is evaLuated step by step, and the resuLt is a state-vaLue pair if the sequence ends with a function invocation. Intermediate structures are composed via the µ operator (see sec. 2.1.).

The correctness checks for operation caLL sequences (i.e. obq ∈ ALgQuaL and opid ∈ opnames(F)) have to be seen in connection with the static semantics as given in [OLt 84].

3.2. VariabLe DecLarations and Assignments

3.2.1. VariabLe DecLarations

VariabLes are aLways decLared having vaLues in specific vaLue sets of aLgebras (TOI). The initiaLization is done impLicitLy (standard objects) or expLicitLy (moduLes).

Sem_5: VariabLe DecLaration

```
 M⟦v: Var⟧ϱσ :=
    Let $(id_1, ..., id_n)$ := (idL v), t := (type v),
          int := (int v)                        in
     Let V := $\varrho(t)\downarrow 2$ in
      Let $Loc_i$ := newLoc($\varrho_{i-1}$), i ∈ (n), $\varrho_0 = \varrho$ in
       if standard(V)↓1
       then Let $\varrho_{n+1} := \varrho_n[id_i \hookleftarrow (Loc_i, t, \varrho_n(t)\downarrow 3)]$,
                $\sigma_{n+1} := \sigma_n[Loc_i \hookleftarrow$ standard(V)↓2], i ∈ (n)in
              $(\varrho_{n+1}, \sigma_{n+1})$
       eLse Let $\varrho_{n+1} := \varrho_n[Loc_i \hookleftarrow (Loc_i, t, \varrho_n(t)\downarrow 3]$,
                $\sigma_{n+1} := \sigma_n[Loc_i \hookleftarrow$ VaL(i)], i ∈ (n)
                   where VaL(i) := Let $\{Lv_1, ..., Lv_n\}$ :=
                                          LocaL variabLes
                                          of t in
                                     Let (ϱ', σ') :=
                                              M⟦int⟧($\xi_n$, $\sigma_n$) in
                         ($\sigma'(\xi'(lv_1)\downarrow 1$, ..., $\sigma'(\xi'(lv_n)\downarrow 1)$)
                                                                      in
         ($\xi_{n+1}$, $\sigma_{n+1}$)
```

This semantic clause also covers implicit type definitions in variable declarations. In that case standard type identifiers are generated.

3.2.2. Assignments

ModPascal extends Pascal the assignment statement in that arbitrary module function calls may occur as left-hand-side structures. Also extended dot notation may be used.

Sem_6: Assignments

```
M⟦a: Assg_stmt⟧$\xi\sigma$ :=
   let v := (assg_var a), e := (expr a) in
   case v ∈ Id :
       let (($\xi_1$, $\sigma_1$), r) := E⟦e⟧$\xi\sigma$          in
        let $\sigma_2$ := $\sigma_1[\xi_1(v)\downarrow 1 \leftarrow r]$,        in
            ($\xi_1$, $\sigma_2$)

   case v ∈ Comp_var = (array_var: Id, exprL: ExprL) v
                          Field_designator :
    case v ∈ (array_var: Id, exprL: ExprL) :
    let aid := (array_var v), ($e_1$, ..., $e_n$) := (exprL v) in
     let ($e_1$', ..., $e_n$') := the evaluated index expressions
     in
      let (($\xi_1$, $\sigma_1$), r) := E⟦e⟧$\xi\sigma$          in
       let $\sigma_2$ := $\sigma_1[\xi_1(aid)\downarrow(e_1', ..., e_n') \leftarrow r]$ in
          ($\xi_1$, $\sigma_1$)

    case v ∈ Field_designator :
     case v ∈ (comp_var: Assg_var, field_id: Id) :
      let $V_{id}$ denote the record variable extractable from
                                        (comp_var v) in
       let fid := (field_id v)   in
        let (($\xi_1$, $\sigma_1$), r) := E⟦e⟧$\xi\sigma$ in
         let $\sigma_2$ := $\sigma_1[\xi_1(V_{id})\downarrow fid \leftarrow v]$    in
            ($\xi_1$, $\sigma_2$)

   case v ∈ (ref_var: Ref_var, field_id: Id) :
    let rid := (ref_var v), fid := (field_id v)      in
     let loc := $\sigma(\xi(rid)\downarrow 1)$    in
      let (($\xi_1$, $\sigma_1$), r) := E⟦e⟧$\xi\sigma$    in
       let $\sigma_2$ := $\sigma_1[loc \leftarrow r]$            in
           ($\xi_1$, $\sigma_1$)

   case v ∈ Op_designator :
    let (($\xi_1$, $\sigma_1$), $r_1$) := E⟦v⟧$\xi\sigma$    in
     let (($\xi_2$, $\sigma_2$), $r_2$) := E⟦e⟧$\xi_1\sigma_1$        in
      let loc := $\xi(r_1)\downarrow 1$    in
```

```
        let $\sigma_3$ := $\sigma_2$[loc ↤ r]     in
            ($\xi_2$, $\sigma_3$)
```

Remark: In Sem_7 assumption 3.1.2.-2 on the side effects on expressions is also involved.

3.3. Type Definitions

We distinguish between the various kinds of type definitions according to their degree of freedom of user supplied parts:

- Standard types are predefined. Their semantics is fixed.
- Standard type generators generate partially predefined algebras. The partiality of the definition concerns either carriers or/and operations. The missing parts are supplied by the users type definition.
- Nonstandard type generators generate algebras for which complete definition has to be supplied by the user.

The semantic concept of algebra is employed for each type definition.

Sem_7: Type Definition

```
M⟦t: Typedef⟧$\xi\sigma$ :=
 let tid := (type_id t), tpe := (type t) in
  if tpe ∈ Id then
     let loc := newloc($\xi$) in
     let $\xi_1$ := $\xi$[tid ↤ (loc, $\xi$(tpe)↓2, $\xi$(tpe)↓3)] in
     let $\sigma_1$ := $\sigma$[loc ↤ $\sigma$($\xi$(tpe)↓1),
                        $\xi$(main)↓1 ↤ $\sigma$($\xi$(main)↓1) ∪ $\sigma$($\xi$(tpe)↓1)]in
           ($\xi_1$, $\sigma_1$)
  else
     (case tpe ∈ Module_type :
              let ((obq, v, a), ($\xi_1$, $\sigma_1$)) := Mm⟦tpe⟧$\xi\sigma$ in
              let ($\xi$, $\sigma$) := ($\xi_1$, $\sigma_1$)          in
      case tpe ∈ Instantiate_type:
              let ((obq, v, a), ($\xi_1$, $\sigma_1$)) := Mi⟦tpe⟧$\xi\sigma$ in
              let ($\xi$, $\sigma$) := ($\xi_1$, $\sigma_1$)          in
      otherwise let (obq, v, a) := M⟦tpe⟧$\xi\sigma$  in)
    let loc := newloc($\xi$)                      in
     let $\xi_1$ := $\xi$[tid ↤ (loc, obq, v)]           in
      let $\sigma_1$ := $\sigma$[loc ↤ a,
                      $\xi$(main)↓1 ↤ $\sigma$($\xi$(main)↓1 ∪ a]    in
           ($\xi_1$, $\sigma_1$)
```

To evaluate non-standard type definitions the semantic functions Mm and Mi are used. The reason is that the operations of the new structure are installed not only in the algebra but also in the environment (i.e. ξ(id) is defined for module operations id). To enable this the defining environment has to be passed to the type definition clause.

3.3.1. Standard Types

The standard types of ModPascal are BOOLEAN, INTEGER, REAL, and CHAR. The semantics of each is implementation dependant. We define their meaning via the standard algebras of 2.2.2.

```
Sem_8: Standard Types

  Mt⟦BOOLEAN⟧ξσ := (BOOLEAN, TOI(BOOL-Alg), BOOL-Alg)
  Mt⟦INTEGER⟧ξσ := (INTEGER, TOI(INT-Alg), INT-Alg)
  Mt⟦REAL⟧ξσ := (REAL TOI(REAL-Alg), REAL-Alg)
  Mt⟦CHAR⟧ξσ := (CHAR, TOI(CHAR-Alg), CHAR-Alg)
```

Since they are predefined and language-inheritant, the semantics of standard types is state independant.

3.3.2. Standard Type Generators

The standard type generators of ModPascal are patterns, which expose a semantic fragment that has to be completed by user supplied information. The kind of information is generator dependant. Standard type generators are the following: scalar, subrange, array, record, set, file, pointer.

(a) Scalar Types

The semantic algebra for scalars includes standard operations as 'succ', 'pred' or '<=' and INT-Alg. The missing information is a value set, and it is supplied in the type definition. The algebra pattern is:

```
Sc-Alg

 Signature Σ-Sc := (OB-Sc, OP-Sc) with
           OB-Sc := {scalar, integer, boolean}
           OP-Sc := {pred, succ, ord, chr, <, >, <=, >=,
                     <>, =} ∪ OP-I
           arity-Sc : OP-Sc \ OP-I ──> (OB-Sc* x OB-Sc)
           (e.g. arity-Sc(<=) := (scalar scalar, boolean))

 Then SC-Alg := (C-Sc, F-Sc) ∈ Alg[Σ-Sc] with
        C-Sc := {Sc-Val, I-Val, B-Val}, Sc-Val := {<user>}
        F-Sc := {pred, succ, ord, chr, <, >, <=, >=, =, <>}
                 ∪ F-I
              where ord(x) := <user>
                         ...
                        etc.
 TOI(SC-Alg) := Sc-Val
```

<u>Remark</u>: The functions of F-Sc \ F-I are ambiguously denoted by the function names of OP-Sc \ OP-I.

This semantic algebra pattern is updated in the elaboration of the scalar type definition. The indication <user> in the Sc-Alg pattern defines the points at which the information extracted from the user supplied type definition is built in.

```
Sem_9: Scalar type

Mt⟦sc: Scalar_type⟧ξσ :=
  let (id$_1$, ..., id$_n$) := (idL sc)   in
    if ∃ i ∈ (n) . ξ(id$_i$) = ⊥ then error else
      let Sc-Val := {id$_1$, ..., id$_n$, ⊥}   in
       let ord: Sc-Val —> I-Val with ord(id$_i$) = i   in
          (SCALAR, Sc-Val, Sc-Alg)
```

Remark: All id$_i$ are installed in (ξ, σ) with value "id$_i$" (the string):

```
let loc$_i$ := newloc(ξ), loc$_i$ ≠ loc$_j$, i,j ∈ (n) in
let ξ$_1$ := ξ[id$_i$ ↤ (loc$_i$, CONST, ⊥], i ∈ (n) in
let σ$_1$ := σ[loc$_i$ ↤ "id$_i$"], i ∈ (n) in
     (ξ$_1$, σ$_1$)
```

(b) Subrange Types

Subrange types can be declared upon INTEGER or scalar types. Semantically, the carriers of INT-Alg/Sc-Alg are modified (the TOI is restricted), and the operations have to respect the new boundaries of their arguments.

```
Sub-Alg

case Subrange of INTEGER :
Then SUB-Alg := (C-SUB, F-SUB) ∈ Alg[Σ-I] with
     C-SUB := {SUB-Val, B-Val}, SUB-Val := {<user>}
     F-SUB := F-I but functions evaluate to ⊥ if arguments
              or result are out of range

case Subrange of scalar type :
Then SUB-Alg := (C-SUB, F-SUB) ∈ Alg[Σ-Sc] with
     C-SUB := {SUB-Val, B-Val}, SUB-Val := {<user>}
     F-SUB := F-Sc but functions evaluate to ⊥ if arguments
              or result are out of range

anycase
          TOI(SUB-Alg) := SUB-Val
```

Remark: Both variants of SUB-Alg are based on the associated signatures Σ-I and Σ-Sc resp. and therefore are contained in Alg[Σ-I] and Alg[Σ-Sc] resp.

```
Sem_10: Subrange Type

Mt⟦st: Subrange_type⟧ξσ :=
  let l := (lower st), u := (upper st)   in
```

```
case l, u ∈ I-Val :
   if not (l < u) then error else
    let SUB-Val := {l, l+1, ..., u-1, u}   in
     let F-SUB := {f| f ∈ F-I, but f evaluates to ⊥, if
                      arguments or result ∉ SUB-Val} in
          (SUBRANGE, SUB-Val, SUB-Alg)

case l, u ∈ Sc-Val :
   if not (l < u) then error else
    let {$v_1$, ..., $v_n$} := {v| v ∈ Sc-Val, $v_1$ = l, $v_n$ = u}
      with ord($v_i$) = ord($v_{i-1}$)+1, i ∈ {2, ..., n}  in
     let SUB-Val := {$v_1$, ..., $v_n$}   in
      let F-SUB := {f| f ∈ F-Sc, but f evaluates to ⊥,
                  if arguments or result ∉ SUB-Val} in
           (SUBRANGE, SUB-Val, SUB-Alg)
```

Remark: Although subranges copy most of the structure of their basing type, they are viewed as constituting an own algebra. As a consequence, subrange type variables are type inconsistent with their basing type operations. This is contrary to the coercions performed in Pascal at this point.

(c) Array Types

The array algebra pattern lacks two informations: the index type(s) and the component type. The index type(s) (scalars or subranges) constitute selector operations of the algebra. Since components occur, assignment operations are needed.

```
Ar-Alg

Signature Σ-Ar := (OB-Ar, OP-Ar) with
          OB-Ar := {array, component, $index_1$, ..., $index_n$}
          OP-Ar := {read, assign} ∪ OP-Co ∪ OP-$I_1$ ∪ ...
                                            ∪ OP-$I_n$
          arity-Ar: {read, assign} ---> (OB-Ar* x OB-Ar)
          (e.g. arity-Ar(read) := (array $index_1$ ... $index_n$,
                                            compoment))
Then Ar-Alg := (C-Ar, F-Ar) ∈ Alg[Σ-Ar] with
          C-Ar := {Ar-Val, Co-Val, $In_1$-Val, ..., $In_n$-Val}
                  Ar-Val := {($x_1$, ..., $x_n$)| $x_i$ ∈ Co-Val}
                  Co-Val := {<user>}, $In_1$-Val := {<user>},
                        ..., $In_n$-Val := {<user>}
          F-Ar := {read, assign} ∪ F-Co ∪ F-$In_1$ ∪ ...
                                             ∪ F-$In_n$
               where read(a,$i_1$, ..., $i_n$) := <component
                            selected by $i_1$, ..., $i_n$ in a>
                          ...
                         etc.
TOI(Ar-Alg) := Ar-Val
```

Remarks: a) The functions read/assign are ambiguously denoted by the function names read/assign of OP-Ar.

b) Co-Alg, In_1-Alg, ..., In_n-Alg are the algebras for the component and index types resp.
c) C-Ar contains additional value sets, if C-Co, $C\text{-}In_1$, ..., $C\text{-}In_n$ do so. Then Σ-Ar is extended, too.

```
Sem_11: Array type

Mt⟦a: Array_type⟧ξσ :=
  let ($it_1$, ..., $it_n$) := (indexL a), ct := (comp a) in
   let $In_i$-Alg := (M⟦$it_i$⟧ξσ)↓3, i ∈ (n)        in
    let Co-Alg := (M⟦ct⟧ξσ)↓3                          in
      (ARRAY, Ar-Val, Ar-Alg)
```

Remark: If it_i, i ∈ (n) are type identifier of already defined types, then M is applied.

(d) Record Types

The record algebra has to be completed with the indication of the component selectors and types. If variant parts occur the access to components is dependant of values of access control selectors.

```
Re-Alg

Signature: Σ-Re := (OB-Re, OP-Re) with
           OB-Re := (record, $field_1$, ..., $field_n$)
           OP-Re := {$assign_1$, ..., $assign_n$, $read_1$, ...,
                     $read_n$} ∪ $OP\text{-}F_1$ ∪ ... ∪ $OP\text{-}F_n$
           arity-Re: OP-Re \ ($OP\text{-}F_1$ ∪ ... ∪ $OP\text{-}F_n$) ──>
                                                  (OB-Re* x OB-Re)
           (e.g. arity-Re($read_n$) := (record, $field_n$))
Then Re-Alg := (C-Re, F-Re) ∈ Alg[Σ-Re] with
  C-Re := {Re-Val, $Co_1$-Val, ..., $Co_n$-Val}
          Re-Val := {($x_1$, ..., $x_n$) | $x_i$ ∈ $Co_i$-Val, i ∈ (n)}
          $Co_1$-Val := {<user>}, ..., $Co_n$-Val := {<user>}
  F-Re := {$assign_1$, ..., $assign_n$, $read_1$, ..., $read_n$}
          ∪ $F\text{-}Co_1$ ∪ ... ∪ $F\text{-}Co_n$
          where $read_i$(r) := <if the r↓i component is a
                                fixed type or if the variant
                                selector has appropriate
                                value, then r↓i, otherwise ⊥>
                   ...
                  etc.
TOI(Re-Alg) := Re-Val
```

Remarks: a) The functions $read_i$/$assign_i$ are ambiguously denoted by the function names $read_i$/$assign_i$ of OP-Re.
b) Co_i-Alg are the algebras of the field types.
c) C-Re contains additional value sets, if $C\text{-}Co_i$ does

so. Then Σ-Re is extended, too.

d) Variants are treated as ordinary field types. Only the access functions reflect their activeness resp. inactiveness.

```
Sem_12: Record types

Mt⟦r: Record_type⟧ξσ :=
  let ($fp_1$, ..., $fp_n$) := (fixed_partL r),
      ($vp_1$, ..., $vp_m$) := (variant_partL r)               in
  let ($fid_{i1}$, ..., $fid_{in(i)}$) := (idL $fp_i$),
      $ft_i$ := (type $fp_i$), i ∈ (n)              in
  let $tg_i$ := (tag $vp_i$),($v_{i1}$, ... $v_{im(i)}$) :=(variantL $vp_i$)
                                              i ∈ (m) in
  let $cl_{ij}$ := (constL $v_{ij}$),
          $vt_{ij}$ := $\mu_0$(fixedL: (fixedL $v_{ij}$),
                      (variant_partL: (variant_partL $v_{ij}$)),
                                 i ∈ (n), j ∈ ($m_i$)     in
  let (RECORD, $Re\text{-}Val_{ij}$, $Re\text{-}Alg_{ij}$) := Mt⟦$vt_{ij}$⟧ξσ,
   {$Re\text{-}Val_{ij}$, $Co_{ij1}$-Val, ..., $Co_{ijk(i)}$-Val} := ($Re\text{-}Alg_{ij}$)↓1
   {$assign_{ij1}$, ..., $assign_{ijk(i)}$, $read_{ij1}$, ..., $read_{ijk(i)}$}
     := ($Re\text{-}Alg_{ij}$)↓2 \ ($F\text{-}Co_{ij1}$ ∪ ... ∪ $F\text{-}Co_{ijk(i)}$),
                                       i ∈ (m),  j ∈ ($m_i$) in
              n              m     $m_i$
  let $z_1$ :=  Σ $n_i$, $z_2$ :=  Σ     Σ  $\mathcal{C}$(i, j), z := $z_1$+$z_2$
             j=1            i=1   j=1
       where $\mathcal{C}$(i, j) = a: ⟺ $Re\text{-}Val_{ij}$ ={($x_1$, ..., $x_a$)|
                            $x_k$ ∈ $Co\text{-}Val_{ijk}$, k ∈ (a)} in
                            u-1
  let C1 ≡ (1≤t≤$z_1$ and t=  Σ $n_i$+w, w ∈ ($n_u$))
                            i=1
                             i-1 $m_i$         j-1
      C2 ≡ ($z_1$≤t≤z and t =  Σ  Σ $\mathcal{C}$(x,y) + Σ $\mathcal{C}$(i,y)+k
                            x=1y=1         y=1   in
                       ┌ (M⟦$ft_u$⟧ξσ)↓2 if C1 holds
  let $Co_t$-Val :=   │                                          in
                       └ $Co_{ijk}$-Val    if C2 holds

  let F-Re := {$assign_{ijk}$, $read_{ijk}$| i ∈ (m), j ∈ ($m_i$),
                                              k ∈ $\mathcal{C}$(i, j)}
                     m    $m_i$  $\mathcal{C}$(i,j)
                 ∪   U    U     U      $F\text{-}Co_{ijk}$            in
                    i=1  j=1   k=1

  case $tg_i$ ∈ (tagid: Id, typeid: Id) :
      let $tgid_i$ := (tagid $tg_i$), $tgtp_i$ := (typeid $tg_i$)
       let $Co_{z+i}$-Val := (M⟦$tgtp_i$⟧ξσ)↓2          in
        if not ($cl_{ij}$ ⊆ $Co_{z+i}$-Val) then error else
          F-Re := F-Re ∪ {$assign_{z+i}$, $read_{z+i}$} where
                 <$assign_a$/$read_a$, $z_1$ < a < z are modified:
                  let a satisfy C2 in
                   if $read_{z+i}$(r) ∈ $cl_{ij}$ then
                       <normal evaluation>
                    else error >                                in
          (RECORD, Re-Val, Re-Alg)
```

```
case $tg_i$ ∈ (typeid: Id) :
     (RECORD, Re-Val, Re-Alg)
```

Remarks: a) Sem_12 generates a structure for Re-Val that encloses a component position for all field types (in appropriate number and independant of definition inside a variant) as well as for all tag field types (Co$_{z+i}$-Val). This emphasizes the semantic importance of tag types which is generally treated superficially ([ISO 7185], [SIEM 83]).

b) The semantics of records makes nested structures 'flat' by providing separate access operations for every occurring field.

c) Occurring variants are treated different depending on their tag field: checks of activeness are only performed, if a tag field selector is defined. Some Implementations create the active instance of free variants (no tag field selector) after the first occurrence of a selector evaluation belonging to the variant. Lateron, no other variants are activatable.

(e) Set Types

The set algebra has to be completed only by indication of a base scalar or subrange type.

```
Set-Alg

Signature Σ-Set := (OB-Set, OP-Set)   with
          OB-Set := {set, component, boolean}
          OP-Set := {+, -, *, =, <>, <=, >=, IN}
                    ∪ OP-Co ∪ OP-B
          arity-Set: OP-Set \ (OP-Co ∪ OP-B) ──> (OB-Set* x
                                                      OB-Set)
          (e.g. arity-Set(IN) = (component set, boolean))

Then Set-Alg := (C-Set, F-Set) ∈ Alg[Σ-Set] with
       C-Set := {Set-Val, Co-Val, B-Val}
       Set-Val := ℘(Co-Val), Co-VAL := {<user>}
       F-Set := {+, -, *, =, <>, <=, >=, IN} ∪ F-Co ∪ F-B
                where +(s, t) := {x| x ∈ s or x ∈ t}
                         ...
                         etc.
TOI(Set-Alg) := Set-Val
```

Remarks: a) The functions in F-Set \ (F-Co ∪ F-B) are ambiguously denoted by the function names of OP-Set \ (OP-Co ∪ OP-B).

b) The restriction on the component type to be scalar or subrange type is due to the fact that sets are represented as bit vectors of at most machine word size in many implementations. This again imposes a maximal cardinality on the component type.

```
Sem_13: Set types

Mt⟦s: Set_type⟧ξσ :=
   case s ∈ Scalar_type :
      let (SCALAR, Sc-Val, Sc-Alg) := Mt⟦s⟧ξσ  in
       let Co-Val := Sc-Val, F-Co := F-Sc  in
           (SET, Set-Val, Set-Alg)
   case s ∈ Subrange_type :
       let (SUBRANGE, Sub-Val, Sub-Alg) := Mt⟦s⟧ξσ  in
        let Co-Val := Sub-Val, F-Co := F-Sub   in
           (SET, Set-Val, Set-Alg)
   case s ∈ Id :
      let (T, T-Val, T-Alg) := (ξ(s)↓2, ξ(s)↓3, σ(ξ(s)↓1))
       if T = SCALAR then
                          let Co-Val := Sc-Val, F-Co := F-Sc
                              in (SET, Set-Val, Set-Alg)
       elseif T = SUBRANGE then let Co-Val := Sub-Val,
                                         F-Co := F-SUB       in
                                     (SET, Set-Val, Set-Alg)
       else error
```

Remarks: a) The first two cases reflect implicit types as component types.
b) The implementation dependant cardinality of Co-Val is disregarded.

(f) File Types

The file algebra is fixed up to the indication of the file components.

```
Fi-Alg

Signature: Σ-Fi := (OB-Fi, OP-Fi)  with
           OB-Fi := {file, component, boolean}
           OP-Fi := {put, get, reset, rewrite, eof}
                    ∪ OP-Co ∪ OP-B
           arity-Fi: OP-Fi \ (OP-Co ∪ OP-B) ——>
                                            (OB-Fi* x OB-Fi)
           (e.g. arity-Fi(reset) := (file, file))
Then Fi-Alg := (C-Fi, F-Fi) ∈ Alg[Σ-Fi] with
     C-Fi := {Fi-Val, Co-Val, B-Val}
        Fi-Val := {((x_1, ..., x_n), c, b, j)| x_i ∈ Co-Val,
                  i,j ∈ (n), n ∈ ℕ, c ∈ Co-Val, b ∈ B-Val}
        Co-Val := {<user>}
     F-Fi := {put, get, reset, rewrite, eof}
             ∪ F-Co ∪ F-B
             where put(f,c) := <if f is in generation mode,
                                 then c is appended,
                                 otherwise ⊥ >
           ...
```

```
                 etc.
 TOI(Fi-Alg) := Fi-Val
```

Remarks: a) File incarnations are viewed as quadruples: the sequence of elements, a special file communication component, an indicator for the current mode, and a pointer to the actual position during inspection mode.

b) The special file communication component play the role of the file variable. It is modified or examined by the file operations. The explicit modification by assignment has to be modelled by addition of an assignment operation and a new carrier 'state' that keeps track of side-effects of operations (see (d) and remark b) of Sem_14).

Sem_14: File types

```
 Mt⟦ft: File_type⟧ξσ :=
   Let t := (type ft) in
    Let Co-Val := (M⟦t⟧ξσ)↓2 in
         (FILE, Fi-Val, Fi-Alg)
```

Remarks: a) Since only type identifier are allowed for component types, ξ(t)↓2 also selects Co-Val.

b) Assignment to the file variable can be described as follows:
Let f denote a file, and f↓ the file variable.

```
M⟦f↓ := exp⟧ξσ :=
     Let (s, c, b, j) := ξ(f)↓1 in
     Let c' := E⟦exp⟧ξσ in
     Let σ_1 := σ[ξ(f)↓1 ↤ (s, c', b, j)] in
          (ξ, σ_1)
```

(g) Pointer Types

Pointer types play a special role in ModPascal (and Pascal). They are the only structures whose incarnations refer by definition to hardware properties (memory addresses and contents). Also they allow different incarnations point to the same memory cell such that an implicit value change of a pointer type variable is possible even if no assignment to it occurs.

This behaviour could be modelled in algebraic terms, but only with great struggles. Since all side-effects of the above kind can be administrated in 'states', only a new (abstract) sort 'state' has to be added to all signatures, operation arities and algebras. Then the algebraic description would show state transformation properties similar to a denotational semantics. This introduces complexity in the pointer type description, and for consistency reasons, in the whole treatment up to now, since all structures have to be reformulated.

This task is skipped in this paper, partLy because onLy tedious work is associated to it that does not provide new insights into ModPascaL characteristic features, partLy because aLgebraic descriptions invoLving 'state' sorts are not of great interest of current abstract data type theory research (they are too 'concrete'). NevertheLess, this shouLd be understood onLy as postponement, and a comprehensive and compLete semantics of ModPascaL wiLL incLude pointer type semantics.

3.3.3. Non-Standard Type Generators

There are two non-standard type generators in ModPascaL: moduLe types (sec. 3.4.) and instantiate types (sec. 3.7.). They differ from standard types in that aLL information necessary to buiLd their semantic aLgebra is extracted from the type definition, i.e. there is no semantic frame with hoLes to be fiLLed.

3.4. ModuLe types

A moduLe type definition introduces types as weLL as operations in arbitrary number. This fact forbids an anaLogous formaLization of the semantics as for standard object definitions.
ModuLe operation decLarations differ from ordinary operation decLarations in

- operation header and body are dispart
- occurrences of gLobaL variabLes are restricted to the LocaL variabLe set of the moduLe
- occurrences of moduLe operation caLLs are resticted to visibLe objects, where visibiLity is induced by the use-reLation of the moduLe type definition (see aLso [OLt 84])

A moduLe type object possesses a moduLe state. It consists of the vaLues of LocaL variabLes, and is onLy accessabLe by the operations defined in the associated definition. Procedures modify the state, functions extract information from it without changes, and initiaLs suppLy first vaLues.

Sem_15: ModuLe type

```
Mt⟦m: ModuLe_type⟧ϱσ :=
 Let ($u_1$ ∧ ..., $u_a$) := (useL m),
     ($p_1$, ..., $p_b$) := (pubLicL m),
     ($lt_1$, ..., $lt_c$) := (LocaL_typeL (LocaL m)),
     ($lv_1$, ..., $lv_d$) := (LocaL_varL (LocaL m)),
     ($lo_1$, ..., $lo_e$) := (LocaL_operationL (LocaL m)),
     ($o_1$, ..., $o_f$) := (operationL m)
                                                      in
 Let U := $\bigcup_{i \in (a)} \sigma(\varrho(u_i)\downarrow 1)$     in
 Let ($\varrho_0$, $\sigma_0$) := ($\varrho$, $\sigma$)                  in
```

```
let loc_i := newloc(ρ_i)
    where (case p_i ∈ Proc_head :
                 let opid_i := (proc_id p_i), obq_i := PROC      in
           case p_i ∈ Func_head :
              let opid_i := (func_id p_i), obq_i := FUNC,
                     res_i := (result p_i)                        in
           case p_i ∈ Init_head :
              let opid_i := (init_id p_i), obq_i := INIT         in
            , i ∈ (b))
           ρ_{i+1} := ρ_i[opid_i ↤ (loc_i, obq_i, if obq_i = FUNC
                                                    then res_i
                                                    else ⊥)],
           σ_{i+1} := σ[loc_i ↤ ⊥], i ∈ (b)                      in
let (ρ_0, σ_0) := (ρ_b, σ_b)                                     in

let loc_i := newloc(ρ_i), i ∈ (c)
    where ρ_{i+1} := ρ_i[(typeid lt_i) ↤ (loc_i,
                                  (Mt⟦(type lt_i)⟧ρ_iσ_i)↓1,
                                 (Mt⟦(type lt_i)⟧ρ_iσ_i)↓2)],
          σ_{i+1} := σ_i[loc_i ↤ (Mt⟦(type lt_i)⟧ρ_iσ_i)↓3],
            i ∈ (c)                                               in
let (ρ_0, σ_0) := (ρ_c, σ_c)                                     in

let (ρ_{i+1}, σ_{i+1}) := M⟦lv_i⟧ρ_iσ_i, i ∈ (d)                 in
let LV := U  (idL lv_i)                                           in
        i∈(d)
let (ρ_0, σ_1) := (ρ_d, σ_d)                                     in

let loc_i := newloc(ρ_i)                                          in
    where (case lo_i ∈ Proc_head :
                let opid_i := (proc_id lo_i), obq_i := PROC      in
           case lo_i ∈ Func_head :
              let opid_i := (func_id lo_i), obq_i := FUNC,
                     res_i := (result lo_i)                       in
            , i ∈ (e))
           ρ_{i+1} := ρ_i[obid_i ↤ (loc_i, obq_i, if obq_i = FUNC
                                                      then res_i
                                                   else ⊥)],
           σ_{i+1} := σ_i[loc_i ↤ ⊥], i ∈ (e)                    in
let (ρ_0, σ_0) := (ρ_e, σ_e)                                     in

let LV := U  (idL lv_i)                                           in
        i∈(d)
(case o_i ∈ Proc_spec :
 let opid_i := (proc_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
     D_i := (LV ∪ U  (idL pl_j)) x LV                            in
                 j∈(g)
 case o_i ∈ Func_spec :
 let opid_i := (func_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
     D_i := (LV ∪ U  (idL pl_j)) x LV x ρ(opid_i)↓3              in
                 j∈(g)
 case o_i ∈ Init_spec :
 let opid_i := (init_id o_i), (pl_1, ..., pl_g):=(paramL o_i),
     D_i := (LV ∪ U  (idL pl_j)) x LV                            in
                 j∈(g)
        $, i \in (f))$
  Let $(ST_1, ..., ST_f) := fix\ T_1, ..., T_f . \lambda\xi_1\sigma_1$ .
     $(M[\![(body\ o_1)]\!]\xi_1[opid_1 \leftarrow\!\shortmid (\xi(opid_1)\downarrow 1, \xi(opid_1)\downarrow 2, \perp)]$
                    $\sigma_1[\xi(opid_1)\downarrow 1 \leftarrow\!\shortmid \mathcal{R}(\{T_1, ..., T_f\}, \xi_1, D_1)],$
     ...
     $M[\![(body\ o_f)]\!]\xi_1[opid_f \leftarrow\!\shortmid (\xi(opid_f)\downarrow 1, \xi(opid_f)\downarrow 2, \perp)]$
                    $\sigma_1[\xi(opid_f)\downarrow 1 \leftarrow\!\shortmid \mathcal{R}(\{T_1, ..., T_f\}, \xi_1, D_f)])$
                                                                        in
  Let $opdef_i := \mathcal{R}(ST_i, \xi_i, D_i), i \in (f)$                                 in
  Let $(\xi_1, \sigma_1) := (\xi_0[opid_i \leftarrow\!\shortmid (\xi(opid_i)\downarrow 1, \xi(opid_i)\downarrow 2, \perp]$
                      $\sigma_0[\sigma_0(\xi_0(opid_i)\downarrow 1) \leftarrow\!\shortmid opdef_i]),$
                      $i \in (f)$                                          in

  Let $M\text{-}Val := X \{\xi_1(id)\downarrow 3 \mid id \in LV\}$                              in
  Let $M\text{-}F := \{\sigma_1(\xi_1(opid_i)\downarrow 1) \mid i \in (f)\}$                         in
  Let $M\text{-}Alg := (\{M\text{-}Val\}, M\text{-}F) \cup U \cup \{\xi_1(typeid\ lt_i)\downarrow 1 \mid$
                                                       $i \in (c)\}$   in
     $((MODULE, M\text{-}Val, M\text{-}Alg), (\xi_1, \sigma_1))$
```

Remarks: a) The explicit binding of module operations in environments has only technical reasons (application of the fixpoint operator). It would suffice to install them directly as algebra functions; see also Sem_3 and Sem_4.

b) The resulting algebra is built on the union of the used ones and equipped with the carrier generated from the cartesian product of the local variable TOI's and with all public and local operations.

c) The semantics of the operations are computed by parallel fixpoint abstraction. By using the operator $\mathcal{R}$ the fixpoint is an algebra function defined on TOI's of local variable and parameter types. The state (ξ_1, σ_1) is assumed to contain the appropriately called and passed formal parameter values.

d) Beside the module algebra, a resulting state is passed to save all parts of the definition. This makes conveniant access in semantic clauses possible that are based on modules (e.g. enrichments, instantiations).

e) TOI(m) := M-Val

3.5. Enrichments

Enrichments are very similiar to module types in that they introduce operations that are only invokable on specific instances of structures. Therefore operations introduced in an enrichment definition are called under the same rules. The main differences to module type definitions are:

- enrichments do not introduce a new type (algebra); as a consequence no variables may be declared of enrichment structures
- operations introduced by an enrichment are uniquely connected to one or more modules.

Enrichments may be seen as an enlargement of sets of operations of already defined algebras. Thus the programmer is enabled to modify an existing structure according to his needs without redefining and renaming. The enlarged structure is made visible through its occurrence in the use clause of a module or an enrichment, and the enrichment operations can be called inside the scope of the using structure.

In Sem_16 the syntactical operator
AO: Public x Enrich_def ──> Id
is used. AO maps a public operation header $p \in$ (publicL a), $a \in$ (addL e), $e \in$ Enrich_def to that object identifier that is enlarged by the occurrence of p in its associated addpart of e:

```
AO(p, e) := ι id ∈ Id .
                 let {$a_1$, ..., $a_n$} := (addL e) in
                   ∃ i ∈ (n) . id = (add_id $a_i$) and
                               p ∈ (publicL $a_i$)
```

Sem_16: Enrichment definition

```
Me⟦e: Enrich_def⟧ϱσ :=
  let eid := (enr_id e), ($u_1$, ..., $u_a$) := (useL e),
      ($a_1$, ..., $a_b$) := (addL e),
      ($o_1$, ..., $o_c$) := (operationL e)  in
  let $aid_i$ := (add_id $a_i$), i ∈ (b)   in
  let ($p_{i1}$, ..., $p_{ib(i)}$) := (publicL $a_i$), i ∈ (b) in
  let ($ϱ_{00}$, $σ_{00}$) := (ϱ, σ)                        in
  let $loc_{ij}$ := newLoc($ϱ_{ij}$)
       where
         (case $p_{ij}$ ∈ Proc_head :
             let $opid_{ij}$ := (proc_id $p_{ij}$), $obq_{ij}$ := PROC,
              AO($opid_{ij}$, e) := $aid_i$                          in
          case $p_{ij}$ ∈ Func_head :
             let $opid_{ij}$ := (func_id $p_{ij}$), $obq_{ij}$ := FUNC,
              AO($opid_{ij}$, e) := $aid_i$, $res_{ij}$:= (result $p_{ij}$)in
          case $p_{ij}$ ∈ Init_head :
             let $opid_{ij}$ := (init_id $p_{ij}$), $obq_{ij}$ := INIT,
              AO($opid_{ij}$, e) := $aid_i$                          in
         , i ∈ (b), j ∈ ($b_i$))
         $ϱ_{i,j+1}$ := $ϱ_{ij}$[$opid_{ij}$ ↤ ($loc_{ij}$, $obq_{ij}$,
                                       if $obq_{ij}$ = FUNC
                                       then $res_{ij}$ else ⊥)]
         $σ_{i,j+1}$ := σ[$loc_{ij}$ ↤ ⊥], i ∈ (b), j ∈ ($b_i$) in

  let ($ϱ_0$, $σ_0$) := ($ϱ_{bb(b)}$, $σ_{bb(b)}$)    in
  let ($ϱ_{i+1}$, $σ_{i+1}$) := M⟦(paramL $o_i$)⟧$ϱ_i σ_i$, i ∈ (c) in

  let ($ϱ_0$, $σ_0$) := ($ϱ_c$, $σ_c$)    in
  (case $o_i$ ∈ Proc_spec :
     let $opid_i$ := (proc_id $o_i$),
         ($pl_1$, ..., $pl_g$) := (paramL $o_i$),
         $LV_i$ := local variables of AD($opid_i$),
         $D_i$ := ($LV_i$ ∪  ⋃  (idL $pl_j$)) x $LV_i$            in
                          j∈(g)
```

```
    case $o_i \in$ Func_spec :
      Let $opid_i$ := (func_id $o_i$),
          ($pl_1$, ..., $pl_g$) := (paramL $o_i$),
          $LV_i$ := local variables of AD($opid_i$),
          $D_i$ := ($LV_i \cup \bigcup_{j \in (g)}$ (idL $pl_j$)) x $LV_i$ x $\varrho_0(opid_i)\downarrow 3$ in
    case $o_i \in$ Init_spec :
      Let $opid_i$ := (init_id $o_i$),
          ($pl_1$, ..., $pl_g$) := (paramL $o_i$),
          $LV_i$ := local variables of AD($opid_i$),
          $D_i$ := ($LV_i \cup \bigcup_{j \in (g)}$ (idL $pl_j$)) x $LV_i$          in
    , $i \in (c)$)

  Let ($ST_1$, ..., $ST_c$) := fix $T_1$, ..., $T_c$ . $\lambda\varrho_1\sigma_1$ .
     (M⟦body $o_1$)⟧$\varrho_1$[$opid_1 \mapsfrom (\varrho(opid_1)\downarrow 1, \varrho(opid_1)\downarrow 2, \bot)$]
                  $\sigma_1$[$\varrho(opid_1)\downarrow 1 \mapsfrom \mathcal{R}(\{T_1, ..., T_c\}, \varrho_1, D_1)$],
      ...
     M⟦body $o_c$)⟧$\varrho_1$[$opid_c \mapsfrom (\varrho(opid_c)\downarrow 1, \varrho(opid_c)\downarrow 2, \bot)$]
                  $\sigma_1$[$\varrho(opid_c)\downarrow 1 \mapsfrom \mathcal{R}(\{T_1, ..., T_c\}, \varrho_1, D_c)$])
                                                                        in
  Let $opdef_i$ := $\mathcal{R}(ST_i, \varrho_i, D_i)$, $i \in (c)$                       in
  Let ($\varrho_1$, $\sigma_1$) := ($\varrho_0$[$opid_i \mapsfrom (\varrho(opid_i)\downarrow 1, \varrho(opid_i)\downarrow 2, \bot$]
                        $\sigma_0$[$\sigma_0(\varrho_0(opid_i)\downarrow 1) \mapsfrom opdef_i$]),
                        $i \in (c)$                                     in
  Let U := $\bigcup_{i \in (a)} \sigma_1(\varrho_1(u_i)\downarrow 1)$   in
  Let E-F := {$\sigma_1(\varrho_1(opid_i)\downarrow 1)$ | $i \in (c)$}  in

  Let loc := newloc($\varrho_1$)  in
  Let A := U $\cup$ ($\emptyset$, E-F)  in
  Let $\varrho_2$ := $\varrho_1$[eid $\mapsfrom$ (loc, ENRICHMENT, $\bot$)]  in
      $\sigma_2$ := $\sigma_1$[loc $\mapsfrom$ A,
               $\varrho(main)\downarrow 1 \mapsfrom \sigma(\varrho(main)\downarrow 1) \cup A$]  in

      ($\varrho_2$, $\sigma_2$)
```

Remarks: a) The semantics exclude the case of enrichments of standard types with initial operations (see also [Olt 84]).

b) The installation of the new object in the resulting state and the updating of the main program algebra is done by Me explicitly.

c) Enrichments do not possess a type-of-interest, since they are enlargements of several objects with several types-of-interest. Therefore the $\varrho(eid)\downarrow 3$ component is assigned to $\bot$.

3.6. Instantiations

The instantiation construct of ModPascal is employed by a powerfull parameterization mechanism for types and enrichments. Together with the instantiate type definition (see sec. 3.7.) which is used to generate the structure described by instantiation objects it is possible to parameterize each type in a very flexible way. It is not necessary to declare substructures of a type as formal parameters that have to be actualized (generics of ADA require this [ADA 80]); every substructure is a legal formal parameter, and not earlier than in the instantiate type definition itself it is realized which substructures are parameters that have to be actualized, and which are not.

To avoid misunderstandings the ModPascal parameterization concept for types does not enable dynamic parameterization, i.e. run-time parameterization. This lacks support of nearly every existing Pascal compiler (see also sec. 4.), and a comfortable static parameterization feature covers already many practical applications.

3.6.1. Hierarchical Structures and Morphisms

Up to now we had no necessity to take hierarchical structures on sets of ModPascal objects into consideration. For example the use list of a module definition induces a hierarchy on module and enrichment objects. The context-sensitive conditions attached to the correctness of such a hierarchy are given in [Olt 84]. We did not include them here since

- they were mostly of pure syntactical nature and possessed no state dependant character
- the semantics of the hierarchical structure was computable nevertheless.

The second point is due to the fact that the meaning of the use list of a module is the (algebra) union of the meaning of the list elements (algebras), and algebra union is just a technical process (see sec. 2.2.1.).

Instantiations may also be composed in hierarchies. The hierarchy conditions are the same as for modules or enrichments (as described in [Olt 84]). The semantics of an instantiation object - a signature morphism - has to include the semantics of its used objects. To compute this semantics we need a more specific definition of signature morphism that reflects the hierarchical structure of source and target sets (modules, enrichments), and an operator to unite the single elements of the use clause of an instantiation object. Therefore hierarchical structures will be considered in sections 3.6. and 3.7.

Based on the remarks on the relation between $\xi \in$ Env and the ModPascal data base (sec. 2.2.5.) we will first modify our memory model so that representations of an object will include information about those objects that use it (sec. 3.6.1.1.), then define hierarchical structures (sec. 3.6.1.2.), signature morphisms respecting hierarchical structures (sec. 3.6.1.3.) and finally the semantics of an instantiation object definition (sec. 3.6.1.4.).

3.6.1.1. Extended Domains

To be able to model hierarchies appropriately, we modify our set of domains:

$$\begin{aligned}
&Env = (Id \longrightarrow (Loc \times ObQual \times ValQual \times 2^{Id}))\\
&Map = \mathcal{P}(Id \times Id)\\
&Ar \;\; = Id^{*} \times Id\\
&ValQual = \{C \mid C\text{-}TOI(A)\downarrow 1 \text{ for } A \in Alg\} + Ar\\
&Val = D\_BOOL + D\_INT + Id + Alg + ValQual + OpDen + Map
\end{aligned}$$

Environments now include a fourth component that is designed to keep a set of objects which are used by the current one. This component is also defined for standard objects.

The domain Map provides the semantics for instantiation object: mappings between objects and operations.

The domain Ar (arities) serves as a technical domain to express the functionality of an operation. It is enclosed in ValQual, since this component is currently undefined for operation representations in environments (i.e. if $\varrho(id)\downarrow 2 \in \{FUNC, PROC, INIT\}$ then $\varrho(id)\downarrow 3 = \bot$). From now on it is assumed that $\varrho(id)\downarrow 3$ of operation identifier id contains a tuple (id_1 ... id_n, id_{n+1}), where id_1, ..., id_n represent the names of the operations parameter objects and id_{n+1} the name of its target object. This information is thought to be installed during the elaboration of the operation definition.

Val is extended to express the meaning of instantiation objects in $\sigma \in$ Store.

3.6.1.2. Object Hierarchies

In this section we introduce the notion of an object hierarchy that is adjoined to the hierarchy notion of [RL 84]. We start with technical prerequisites, where Obj denotes the syntactic domain of 2.1.2. We give the definitions without reference to any state on a pure syntactical level. The obvious extentions to dynamic behaviour is sketched at the end of the section.

Def. 3.6.1.2.-1 [object relations]

(a) Let ob $\in$ Obj. Then
 U(ob) := (useL ob)
 denotes the set of used objects.
 Remark: For standard objects the selector useL is implicitly defined.

(b) Let OB $\subseteq$ Obj. Then
 $$U(OB) := \bigcup_{ob \in OB} U(ob)$$

(c) Let ob $\in$ Obj. Then
 $R_u(ob) := \{(ob, ob_1) \mid ob_1 \in U(ob)\}$
 denotes the use relation induced by ob.

(d) Let ob $\in$ Obj. Then
 $\bar{R}_u(ob)$ denotes the least relation with
 1) $(ob_1, ob_2) \in R_u(ob) \Rightarrow (ob_1, ob_2) \in \bar{R}_u(ob)$

2) $(ob_1, ob_2) \in \bar{R}_u(ob) \Rightarrow R_u(ob_2) \in \bar{R}_u(ob)$

$\bar{R}_u$ is called the closure of R

(e) Let ob ∈ Obj. Then

$\bar{U}(ob) := \{ob \mid \exists (ob_1, ob_2) \in \bar{R}_u(ob) . (ob = ob_1 \text{ or } ob = ob_2)\}$

(f) Let OB ⊆ Obj. Then

$R_u(OB) := \bigcup_{ob \in OB} R_u(ob)$

(g) Let Ob ⊆ Obj. Then

$\bar{R}_u(OB) := \bigcup_{ob \in OB} \bar{R}_u(ob)$

(h) Let OB ⊆ Obj. Then

$\bar{U}(OB) := \bigcup_{ob \in OB} \bar{U}(ob)$

(i) Let ob ∈ Obj. Then

$$OPS(ob) := \begin{cases} (publicL\ ob) & \text{if } ob \in Type \\ \bigcup_{a \in (addL\ ob)} (publicL\ a) & \text{if } ob \in Enrich\_def \\ \perp & \text{otherwise} \end{cases}$$

denotes the set of newly introduced operations (with functionalities).

Remark: For standard objects the selector publicL is implicitly defined.

(j) Let OB ⊆ Obj. Then

$OPS(OB) := \bigcup_{ob \in OB} OPS(ob)$

¤

Def. 3.6.1.2.-2 $[P_R, S_R, \bar{P}_R, \bar{S}_R]$

Let OB ⊆ Obj.

1) A function P_R: OB ⟶ $\mathcal{P}(OB)$ defined by

$$P_R(ob) := \begin{cases} \emptyset & \text{if } \forall (ob_1, ob_2) \in \bar{R}_u(OB) .\ ob_2 \neq ob \\ \{ob_1, \ldots, ob_n\} & \text{if } \{(ob_1, ob), \ldots, (ob_n, ob)\} \subseteq \bar{R}_u(OB) \end{cases}$$

is called predecessor function.

2) A function S_R: OB ⟶ $\mathcal{P}(OB)$ defined by

$$S_R(ob) := \begin{cases} \emptyset & \text{if } \forall (ob_1, ob_2) \in \bar{R}_u(OB) .\ ob_2 \neq ob \\ \{ob_1, \ldots, ob_n\} & \text{if } \{(ob, ob_1), \ldots, (ob, ob_n)\} \subseteq \bar{R}_u(OB) \end{cases}$$

is called successor function.

3) A function $\bar{P}_R$: OB ⟶ $\mathcal{P}(OB)$ defined by

$$\bar{P}_R(ob) := \begin{cases} \emptyset & \text{if } P_R(ob) = \emptyset \\ \bar{V}_R(ob_1) \cup \ldots \cup \bar{V}_R(ob_n) \cup V_R(ob) & \text{if } V_R(ob) = \{ob_1, \ldots, ob_n\} \end{cases}$$

is called closure predecessor function.

4) A function $\bar{P}_R$: OB ⟶ $\mathcal{P}(OB)$ defined by

$$\bar{S}_R(ob) := \begin{cases} \emptyset & \text{if } S_R(ob) = \emptyset \\ \bar{S}_R(ob_1) \cup \ldots \cup \bar{S}_R(ob_n) \cup S_R(ob) & \text{if } S_R(ob) = \{ob_1, \ldots, ob_n\} \end{cases}$$

is called closure successor function.

¤

Def. 3.6.1.2.-3 [cycle, cyclefree]
Let $OB \subseteq Obj$, $C \subseteq \bar{R}_u(OB)$, $C = \{(ob_1, ob_1'), ..., (ob_n, ob_n')\}$.
C is called cycle of $\bar{R}_u(OB)$, if

1) $\forall\ i \in \{1, ..., n-1\}\ .\ (ob_{i+1} \in S_R(ob_i')$ and $ob_1 \in S_R(ob_n'))$
2) is minimal with 1)

$CY(OB) := \{C \mid C$ is cycle of $\bar{R}_u(OB)\}$.
$\bar{R}_u(OB)$ is called cyclefree, if $CY(OB) = \emptyset$.

¤

Def. 3.6.1.2.-4 [chain]
Let $OB \subseteq Obj$, $C \subseteq \bar{R}_u(OB)$, $C = \{(ob_1, ob_1'), ..., (ob_n, ob_n')\}$.
C is called chain of $\bar{R}_u(OB)$ if

1) $\forall\ i \in \{1, ..., n-1\}\ .\ ob_{i+1} = ob_i'$
2) $\forall\ i,j \in (n)\ .\ (i \neq j \Rightarrow ob_i \neq ob_j$ and $ob_i \neq ob_j'$ and $ob_i' \neq ob_j$ and $ob_i' \neq ob_j')$

$CH(OB) := \{C \mid C$ is chain of $\bar{R}_u(OB)\}$.

¤

Remarks: a) Chains are prestructures of cycles, i.e. to every cycle there is associated a set of chains.
b) Condition 2) is equivalent to: "no cycles occur".
c) For $ob \in Obj$. $CH(ob) := CH(\{ob\})$

Def. 3.6.1.2.-5 [hierarchy]
Let $OB \subseteq Obj$.
$\bar{R}_u(OB)$ is called hierarchy, if

1) $\bar{R}_u(OB)$ is cyclefree
2) $\forall\ ob_1, ob_2 \in \bar{U}(\bar{R}_u(OB))\ .$
$(P_R(ob_1) = P_R(ob_2) = \emptyset \Rightarrow ob_1 = ob_2)$

¤

Remarks: a) $TOP(OB) := \iota\ ob \in OB\ .\ P_R(ob) = \emptyset$ denotes the unique top element of the hierarchy $\bar{R}_u(OB)$. For $ob \in Obj$. $TOP(ob) := ob$
b) Hierarchies can be represented by acyclic directed graphs.

The definitions of this section can be extended to include state dependancy. In this case we assume a unique association between the syntactic object $ob \in Obj$ and its equally named semantic counterpart contained in a given state (ξ, σ). Also selectors and environment components are assumed to be uniquely associated.

Extension 3.6.1.3.-6
Let $ob \in Obj$, and $(\xi, \sigma) \in State$.
Then $\xi(ob)$ is defined with appropriate properties.

(a) $U(ob)\xi\sigma := \xi(ob)\downarrow 4$
(b) $\bar{U}(OB)\xi\sigma := \bigcup_{ob \in OB} U(ob)\xi\sigma$
(c) $R_u(ob)\xi\sigma := \{(ob, ob_2) \mid ob_2 \in U(ob)\xi\sigma\}$

Analogously the operations $\bar{R}_u(ob)\xi\sigma$, $\bar{U}(ob)\xi\sigma$, $R_u(OB)\xi\sigma$, $\bar{R}_u(OB)\xi\sigma$, $OPS(ob)\xi\sigma$, and $OPS(OB)\xi\sigma$ are defined by exchanging the state-invariant operators by their state depending version. $P_R(ob)\xi\sigma$, $\bar{P}_R(ob)\xi\sigma$, $S_R(ob)\xi\sigma$, $\bar{S}_R(ob)\xi\sigma$ are the state dependant predecessor and successor functions, $CY(OB)\xi\sigma$ the state dependant set of cycles of OB, and the hierarchy definition takes over in the same fashion to the dynamic case.

¤

3.6.1.3. Hierarchy respecting signature morphisms

In the definition of signature morphism above (definition 2.2.1.-2) no care is taken to ensure preservation of structures lying on the source or target of the object mapping. But this is highly unwanted if signature morphisms are applied to hierarchies of ModPascal objects. Then morphisms introducing cycles or unconnected graphs are useless since their involvement in an instantiate type definition leads to incorrect programs.

A second effect of clashed hierarchies is that the upwards interface of objects (the set of all objects and operations provided by an object to another one that uses it) may become inconsistent, i.e. it contains operations of objects that miss completely or are incompatible in the resulting hierarchy. Since each object may incorporate all items of the upwards interface of its used objects this means that non-hierarchy respecting morphisms - when applied in an instantiate type definition - can violate interface conditions and therefore generate erroneous ModPascal code.

To recognize these effects as early as possible, we use the concept of hierarchy respecting signature morphisms.

<u>Notation</u>: Let SM = (f, g) denote a signature morphism.
Let R_f, R_g denote the relations associated to f, g.
$Source(R_i) := \{a \mid (a, b) \in R_i\}$,
$Target(R_i) := \{b \mid (a, b) \in R_i\}$, $i \in \{f, g\}$
$Source(SM) := (Source(R_f), Source(R_g))$
$Target(SM) := (Target(R_f), Target(R_g))$

¤

<u>Def. 3.6.1.3.-1</u> [hr]
Let SM = (f, g) denote a signature morphism.
Let $OB_1 := Source(SM)\downarrow 1$, $OB_2 := Target(SM)\downarrow 1$, $R_i := \bar{R}_u(OB_i)$, $i \in \{1,2\}$, where U is a unique relation on OB_1 and OB_2.
Let $\forall\ ob \in U(R_1) \setminus OB_1\ .\ f(ob) = ob$
Let R_1 denote a hierarchy.
SM is called <u>hierarchy respecting (hr)</u> if
$\forall\ C \in CH(OB_1),\ C = \{(ob_1, ob_2), (ob_2, ob_3), \ldots, (ob_{n-1}, ob_n)\}$.
1) $\forall\ ob \in \bar{S}_R(ob_1)\ .\ (f(ob) = ob$ <u>or</u> $f(ob) \in \bar{S}_R(f(ob_1)))$
2) $\forall\ i \in (n-1)\ .\ \forall\ ob \in \bar{S}_R(ob_{i+1})\ .$
$(f(ob) = ob$ <u>or</u> $f(ob) \in \bar{S}_R(f(ob_i))$

¤

<u>Remarks</u>: a) The first condition guarantees that the successor structure of an object is maintained and that the upwards interface remains consistent. The second condition ensures this for the hierarchy spanned

by the object.

b) The hr property depends only on the object mapping f.

<u>Fact 3.6.1.3.-2</u> Let SM be hr, OB := Source(SM)$\downarrow$1, and F(OB) := {f(ob) | ob $\in$ OB}. Then $\bar{R}_u$(F(OB)) is a hierarchy.

¤

The next technical definition is used for abbrevation in sec. 3.6.2.

<u>Def. 3.6.1.3.-3</u> [SM?]
If A = (f, g) denotes a tuple of mappings analogous to those of the signature morphism definition, then the predicate SM? is defined by

$$\text{SM?}(A) := \begin{cases} \text{true} & \text{if } A \text{ denotes a hr signature morphism} \\ \text{false otherwise} & \end{cases}$$

¤

Signature morphisms can be united if their object and operation mappings are compatible.

In the next definition we assume the situation:

$$\begin{tikzcd} OB_{11} \arrow[r, "f_1"] & OB_{12} & & OB_{21} \arrow[r, "f_2"] & OB_{22} \\ OP_{11} \arrow[r, "g_1"'] & OP_{12} & & OP_{21} \arrow[r, "g_2"'] & OP_{22} \end{tikzcd}$$

SM1 SM2

SM_i is based on the signatures Σ_{ij} = (OB_{ij}, OP_{ij}), i,j $\in$ {1,2}.

<u>Def. 3.6.1.3.-4</u> [SM_1 + SM_2]
Let SM_i = (f_i: OB_{i1} —> OB_{i2}, g_i: OP_{i1} —> OP_{i2}), i $\in$ {1,2} denote signature morphisms.
Then the <u>combination of SM_1 and SM_2</u> (denoted by SM_1 + SM_2 = (f: OB_S —> OB_T, g: OP_S —> OP_T) is defined if

a) $\forall$ ob $\in$ (OB_{11} $\wedge$ OB_{12}) . f_1(ob) = f_2(ob)

b) $\forall$ op $\in$ (OP_{11} $\wedge$ OP_{12}) . g_1(op) = g_2(op) holds.

Then

OB_S := OB_{11} $\vee$ OB_{21}, OB_T := OB_{12} $\vee$ OB_{22}, OP_S := OP_{11} $\vee$ OP_{21}, OP_T := OP_{12} $\vee$ OP_{22}, f(ob) := f_i(ob) if ob $\in$ OB_{1i}, i $\in$ {1, 2}, g(op) := g_i(op) if op $\in$ OP_{1i}, i $\in$ {1, 2}

¤

<u>Notation</u>: SM_1 + ...+ SM_n := (SM_1 + (SM_2 + (... +(SM_{n-1} + SM_n) ...)

<u>Remark</u>: SM_1 + SM_2 := $\perp$ if the requirements of the definition are not met.

If two signature morphisms are hr, their combination may loose this property because source and target are simply united by set union. This process can destroy uniqueness of the TOP-object of the source object hierarchy or introduce cycles if the $\bar{R}_u$ operator is based on different use-relations for an object. In this cases the preconditions for hr are not met.

The next corrollary states conditions under which the hr property is preserved.

Corrollary 3.6.1.3.-5
Let $SM_i = (f_i, g_i)$, $i \in \{1, 2\}$ denote a hr signature morphism.
Let $S_i := Source(SM_i)\downarrow 1$, $T_i := Target(SM_i)\downarrow 1$, $i \in \{1, 2\}$ denote sources and targets of the object mappings.
Let $SM_1 + SM_2 = (f, g)$ denote the combination of f_1 and f_2.
If (1) $\forall$ ob $\in (S_1 \cup S_2 \cup T_1 \cup T_2)$. U(ob) is unique
(2) $\bar{R}_u(S_1 \cup S_2)$ is a hierarchy
Then $S_1 + S_2$ is hr.
Proof: Since every element of $S_1 \cap S_2$ is mapped identically by f_1 and f_2, the hr property of $SM_1 + SM_2$ follows from the hr property of SM_1 and SM_2 and (1), (2) directly.
¤

3.6.2. Instantiation Definition

An instantiation definition introduces a hr signature morphism. This morphism can be used in an instantiate type definition to generate a new object hierarchy or in other instantiation definitions.

An instantiation definition consists of at most four parts:
- a use clause
- an object actualization clause
- a type actualization clause
- an operation actualization clause

The use clause allows the inclusion of already defined signature morphisms. The object and type actualization parts are distinguished because modules as well as enrichments may be actualized, and in the latter case the add objects of the enrichments are explicitly mentioned in the type actualization. Object and type actualization are combined to the signature morphism object mapping, while the operation actualization constitutes the operation mapping.

Sem_17: Instantiation Definition

```
M⟦i: Inst_def⟧ϱσ :=
  let in_id := (inst_id i), (I₁, ..., Iₐ) := (useL i),
      (ob₁, ..., ob_b) := (ob_actL i),
      (t₁, ..., t_c) := (type_actL i),
      (op₁, ..., op_d) := (op_actL i)       in
  let (f, g) := σ(ϱ(I₁)↓1) + ... + σ(ϱ(Iₐ)↓)1) in
   if not (SM?((f, g))) then ⊥
   else
    let F := {((old oᵢ), (new oᵢ))| i ∈ (b)} ∪
             {((old tᵢ), (new tᵢ))| i ∈ (d)}  in
    let G := {((old opᵢ), (new opᵢ))| i ∈ (c)} in
```

```
if not (SM?((F, G))) then ⊥ else
 let SM := (f, g) + (F , G)  in
  if not (SM?(SM)) then ⊥ else
   let loc := newloc(ϱ) in
   let ϱ₁  := ϱ[in_id ↤ (loc, INST, ⊥, ⊥)]
       σ₁  := σ[loc ↤ SM]   in
   let σ₂  := σ₁[σ₁(ϱ₁(main)↓1) ↤ σ₁(ϱ₁(main)↓1) ∪
                   ({source(SM), target(SM)}, {SM})] in
  (ϱ₁, σ₂)
```

<u>Remarks</u>: a) SM? is the predicate to indicate signature morphism property of its argument (see sec. 3.6.1.3.).

b) The arity operator for a signature morphism is defined in $(\varrho, \sigma) \in$ State as follows:

$$\text{arity(id)}\varrho\sigma := \begin{cases} \varrho(\text{id})\downarrow 4 & \underline{\text{if}}\ \varrho(\text{id})\downarrow 2 \in \{\text{PROC}, \text{FUNC}, \text{INIT}\} \\ \perp & \underline{\text{otherwise}} \end{cases}$$

With this definition of arity the predicate SM? is computable for identifiers bound in environments $\varrho \in$ Env. For the general solution to the problem of application of syntactical operators to elements of semantic domains see definition 3.7.4.-1.

c) For consistency and for verification contexts (see [Olt 85]), an algebra of the form above is added to 'main'.

Instantiation definitions enlarge the main program algebra, although they are not involved in one verification context that represents one primary application area of the main program algebra (MPA) concept: the transition from ModPascal to Pascal (see sec. 3.7.). In this context the enlargement of MPA is disregarded since instantiations are pure ModPascal objects, i.e. they have no counterpart in Pascal via the precompiling process. Their semantics can be described as some kind of 'meta-functions' of algebras since their object mapping maps carrier sets to carrier sets (and not elements of carrier sets to elements of carrier sets).

<u>3.7. Instantiate Types</u>

The instantiate type definition provides the ModPascal parameterization mechanism. Parameters are all objects occurring in the source hierarchy of the instantiation except of the top object and the standard object BOOLEAN. Instantiations are applied to an object hierarchy to yield a new hierarchy with possibly implicitly generated objects. This will be always the case if the instantiation does not affect hierarchy levels that lay one upon the other and therefore the intermediate structures are based on objects that are already actualized. The necessity of implicit object generation may be visualized best by an example.

Example 3-1
Consider the module hierarchy

$$\begin{tikzcd}
M_1 \arrow[d] \\
M_2 \arrow[d] \\
M_3 \arrow[d] \\
M_4
\end{tikzcd}$$

and the instantiation

instantiation I is M_4 by M_5;
operations op_4 = op_5 ; instend ;

and the instantiate type definition that employs I:

type $\bar{M}_1$ = instantiate M_1 by I ;

The primary effect of this definition is the substitution of M_4 by M_5 in the M_1 hierarchy. But then M_3 is no longer appropriate since it uses M_4 in its object definition and has possibly occurrences of M_4 operations. So $\bar{M}_3$ is generated as a copy of M_3 with exchanged use list and substituted operation calls. Now the same argument is applicable to M_2, resulting in $\bar{M}_2$, and finally to M_1 to yield to $\bar{M}_1$ as outcome of the instantiate type definition.

The resulting hierarchy

$$\begin{tikzcd}
\bar{M}_1 \arrow[d] \\
\bar{M}_2 \arrow[d] \\
\bar{M}_3 \arrow[d] \\
\bar{M}_5
\end{tikzcd}$$

includes the two implicitly generated objects $\bar{M}_2$ and $\bar{M}_3$. ¤

In the following we will firstly extend the data structures on which the semantic clauses for instantiate type definitions will be based (sec. 3.7.1.). Then auxiliary functions will be introduced to manipulate syntactic objects (sec. 3.7.2.). Thereafter a syntactic process for marking an object hierarchy with substitution flags is defined, and a generation algorithm working on marked hierarchies is presented (sec. 3.7.3.). Finally, the embedding of this definitions in the semantic clauses for the instantiate type definition is given in sec. 3.7.4.

3.7.1. Extended Data Structures

To express the semantic of an instantiate type definition more concise we modify our data structure for the syntactic domain Obj.

```
Obj = Type_def_struct v Enrich_def_struct v Inst_def
Type_def_struct = (type_id: Id, type: Type, map: Map,
                   new: Obj)
Map = 𝒫(Id x Id)
Enrich_def_struct = (enr_id: Id, enr: Enr, map: Map,
                     new: Obj)
```

The domains Type, Enr, Map, Id are unchanged.

The extension of the syntactic domains Type_def and Enr_def allows to define the algorithms that are employed in the instantiate type definition semantics on the syntactic entities.
The syntactic domain Map represents mappings between objects where object identifiers are taken as unique references to them and the uniqueness is valid for sets OB of objects.

3.7.2. Auxiliary Functions

In this section we introduce some functions on syntactic domains that are used in the subsequent definitions.

We assume the operator U defined for all objects ob ∈ Obj such that the derived operators $\bar{R}_v$, $\bar{R}_u$, $\bar{U}$ are meaningfull (see definition 3.6.1.2.-1).

The first definition characterizes lists of objects.

Def. 3.7.2.-1 [admissible objectlist]
Let obL ∈ ObjL.
obL is called admissible if

```
  either (first obL) = $\perp$
  or let m := min{n| (first(rest$^n$ obL)) = $\perp$}  in
     let ob$_i$ := (first(rest$^{i-1}$ obL)), i ∈ (n)  in
     let OB$_i$ := {ob$_1$, ..., ob$_{i-1}$}, i ∈ (n)  in
         ∀ob ∈ U(ob$_i$) . (ob ∈ Stand_type or
                          ob ∈ OB$_i$, i ∈ (n))
```

¤

Remark: Admissability corresponds to 'declaration-before-use'.

To convert sets of objects into lists of objects, a specific operator is defined next.

Def. 3.7.2.-2 [SEQ]
Let OB ⊆ Obj, OB = $(ob_1, ..., ob_n\}$.
Let $(i_1, ..., i_n)$ denote an arbitrary permutation of (1, ..., n) such that obL := μ_0(first: $\{ob_{i(1)}\}$, rest: μ_0(first $\{ob_{i(2)}\}$, ..., rest: μ_0(first: $(ob_{i(n)}\}$, rest: $\perp$) ...) with obL ∈ ObL.
Then the operator SEQ: 𝒫(Obj) ⟶ ObL is defined by

$$\text{SEQ(OB)} := \begin{cases} \text{obL} & \text{if obL is admissable} \\ \perp & \text{otherwise} \end{cases}$$

¤

Remark: SEQ is defined if and only if a permutation $(i_1, \ldots, i_n)$ exists that generates an admissable object sequence.

The effect of mappings defined by instantiations is captured on the syntactical level by substitutions of source objects through target objects and by substitution of source object operations through target object operations.

The next definition gives the syntactical operator for this process.

Def. 3.7.2.-3 [Sub]
Let $OB \subseteq Obj$, $D := \bar{U}(\bar{R}_u(OB))$.
Let $f: D \longrightarrow Obj$ denote a mapping.
Let $OB_1 := \{f(ob) \mid ob \in D\}$.
Let $g: OPS(D) \longrightarrow OPS(OB_1)$ denote a mapping.
Then the substitution Sub(OB) according to f and g in OB is defined by

```
1) $\forall ob \in OB$ .
   $SF(ob) := \{s \mid$ $s \in AD(ob)$ and $(s\ ob) \in Id$ and
                       $s = s_n \ldots s_1$ and $s_n \in \{type\_id, enr\_id\}$
   $SG(ob) := \{s \mid$ $s \in AD(ob)$ and $(s\ ob) \in Id$ and
                       $s = s_n \ldots s_1$ and
                       $s_n \in \{proc\_id, func\_id, init\_id\}$
2) $\forall ob \in OB$ .
   Let $\{s_1, \ldots, s_t\} := SF(ob)$ in
   Let $ob_1 := \mu(\ \ldots(\mu(ob;$ $s_1: \{f(s_1\ ob)\});$
                                  $s_2: \{f(s_2\ ob)\}); \ldots);$
                                  $s_t: \{f(s_t\ ob)\})$ in
   Let $\{s_1', \ldots, s_s'\} := SG(ob)$ in
   Let $ob_2 := \mu(\ \ldots(\mu(ob_1;$ $s_1': \{g(s_1'\ ob)\}); \ldots;$
                                   $s_s': \{g(s_s'\ ob)\})$ in
         $S_1(ob) := ob_2$
3) $Sub(OB) := \{S_1(ob) \mid ob \in OB\}$
```

Notation: $OB<f, g> := Sub(OB)$ according to f and g.

¤

Remarks: a) The substitution is defined on a purely syntactical level, i.e. only identifier (object and operation names) are substituted.
b) Application of the (state dependant) substitution operation see ....
c) If $f = \perp$ or $g = \perp$ then $OB<f, g> := OB$.

The next definition introduces measures for hierarchies and objects occurring there.

Def. 3.7.2.-4 [depth, height]
Let $ob \in (Module\_type \vee Enrich\_def)$, such that $\bar{R}_u(ob)$ is hierarchical.
Let $OB(ob) := \bar{U}(\bar{R}_u(ob))$.

1) The function depth: OB(ob) ---> N is defined by:

$$\text{depth}(ob_0) := \begin{cases} 1 & \underline{\text{if}}\ ob = ob_0 \\ m & \underline{\text{if}}\ m = \min\{n \mid \exists\ C \in CH(ob), \\ & \qquad C = \{(ob_1, ob_2), \ldots, \\ & \qquad\qquad (ob_{n-1}, ob_n)\}, \\ & \qquad ob_1 = ob, ob_n = ob_0\} \end{cases}$$

2) The function height: (Module_type $\cup$ Enrich_def) ---> N is defined by:

$$\text{height}(ob) := \begin{cases} 1 & \underline{\text{if}}\ |OB(ob)| = 1 \\ n & \underline{\text{if}}\ n = \max\{\text{depth}(ob_1) \mid ob_1 \in OB(ob)\} \end{cases}$$

¤

<u>Remarks</u>: a) Depth denotes the length of the 'shortest' way from the TOP to an element ob_0 of the hierarchy. Depth(ob_0) = 1 is equivalent to ob_0 = TOP(OB(ob)).
b) Hight denotes the lenght of the 'longest' way from the TOP to an element of the hierarchy spanned by ob.

The next operator checks if an object set and a mapping are compatible, i.e. if the mapping is applicable to the object set.

<u>Def. 3.7.2.-5</u> [Comp?]
Let OBJECT := (Type_def_struct $\cup$ Enr_def_struct)
Then the operator
Comp?: $\mathcal{P}$(OBJECT) x Map ---> D_BOOL
is defined by:
Comp?(OB, M) = true
: $\Longleftrightarrow$ 1) SM?(M) = true
2) source(M) $\subseteq$ OB
3) $\bar{R}_u$(OB) is hierarchical
4) M(TOP(R_u(OB))) = $\perp$

¤

<u>Remarks</u>: a) Conditions 1) - 3) require that the supplied mapping is a signature morphism whose source objects are contained in a hierarchical object set. This fact will be used in Sem_17.
b) Condition 4) excludes the case that the top element of an object hierarchy is modified by a signature morphism. By this, the parameterization of objects is restricted to the non-top elements of hierarchies.

3.7.3. Marking and Generation

In this section the application of a signature morphism to a specific object set is defined as a syntactical tree transformation process. Two steps are distinguished:

- marking the object hierarchy with those substitutions that have to be performed at each node

- performing the substitutions and generating of objects.

The next definitions introduce a hierarchy traversal and marking algorithm and an object generation algorithm.

We interprete the structures in Type_def_struct and Enrich_def_struct as follows:

$t \in$ Type_def_struct: (type_id t), (type t) as usual
(map t): a set of identifier pairs (old, new) indicating the substitution old ↤ new in (type t)
(new t): indicates that (type_id t) and (type t) are substituted by (type_id (new t)) and (type (new t)). If (map t) $\neq \perp$, the corresponding substitution is performed on (type (new t)).

$e \in$ Enrich_def_struct: (enr_id e), (enr e) as usual.
(map e): a set of identifier pairs (old, new) indicating the substitution old ↤ new in (enr e).
(new e): indicates that (enr_id e) and (enr e) are substituted by (enr_id (new e)) and (enr (new e)). If (map e) $\neq \perp$, the corresponding substitution is performed on (enr (new e)).

<u>Def. 3.7.3.-1</u> [MARK]
Let OBJECT := (Type_def_struct $\cup$ Enr_def_struct)
Let OB_1, OB_2 $\subseteq$ OBJECT, $\bar{R}_u(OB_1)$ hierarchical and
f: $OB_1 \longrightarrow OB_2$ a mapping.
Let $\forall$ ob $\in$ $(OB_1 \cup OB_2)$. ((map ob) = (new ob) = $\perp$).

1) The operator

```
    MARK: $\mathcal{P}$(OBJECT) x Map ---> $\mathcal{P}$(OBJECT)
is defined by
    MARK($OB_1$, f) := Let n := height(TOP($OB_1$)) in
                               MARK1($OB_1$, f, n)
```

2) The operator

```
    MARK1: $\mathcal{P}$(OBJECT) x Map x $\mathbb{N}$ ---> $\mathcal{P}$(OBJECT)
is defined by
    MARK1($OB_1$, f, n) :=
      if n = 1 then $OB_1$ else
      Let {$on_1$, ..., $on_a$} := {ob| ob $\in$ $OB_1$ and
                                          depth(ob) = n} in
      Let $Z_0$ := $OB_1$ in
      Let $Z_{i+1}$ := (case f($ob_i$) $\neq$ $ob_i$ :
                Let {$ob_1$, ..., $ob_b$} := $\bar{P}_R$($ob_i$) in
```

```
        let $Z := Z_i \setminus \{ob_1, \ldots, ob_b, ob_i\}$ in
        let $Z' := Z$
              $\cup \{ob_j' \mid ob_j \in$ Type_def_struct
                 and
                 $ob_j' = \mu_0($type_id: (type_id $ob_j$),
                                type: (type $ob_j$),
                                map: (map $ob_j$) $\cup$
                                       $\{(ob_i, f(ob_i))\}$,
                                new: (new $ob_j$))
                 and
                 $j \in (b)\}$
              $\cup \{ob_j \mid ob_j \in$ Enr_def_struct
                 and
                 $ob_j' = \mu_0($enr_id: (enr_id $ob_j$),
                                enr: (enr $ob_j$),
                                map: (map $ob_j$) $\cup$
                                       $\{(ob_i, f(ob_i))\}$,
                                new: (new $ob_j$))
                 and
                 $j \in (b)\}$
              $\cup \mu_0(sel_1$: ($sel_1$ $ob_i$),
                      $sel_2$: ($sel_2$ $ob_i$),
                      map: (map $ob_i$) $\cup$
                                 $\{(ob_i, f(ob_i))\}$,
                      new: $\{f(ob_i)\}$)
              where (case $ob_i \in$ Type_def_struct:
                      $sel_1$ := type_id, $sel_2$ := type
                        case $ob_i \in$ Enr_def_struct :
                      $sel_1$ := enr_id, $sel_2$ := enr)
                                                         in
                 Z'

          case $f(ob_i) = ob_i$ : $Z_i$,

                 $i \in (a)$)       in

               MARK1($Z_a$, f, n-1)
```

¤

<u>Remarks</u>: a) Each element in the hierarchy is marked with the substitutions that have to be performed on it. The marking is performed bottom-up and by exchanging objects through appropriate constructed new ones.

b) The cardinality of OB_1 is not changed.

c) The case $f(TOP(OB_1)) \neq TOP(OB_1)$ is disregarded since this does not correspond to parameterization of types. In that the operator is assumed to evaluate errorneous.

To remove marks from the map component of an object, the operator DEMARK can be applied.

Def. 3.7.3.-2 [DEMARK]
Let OBJECT := (Type_def_struct $\cup$ Enr_def_struct).
Let OB_1, OB_2 $\subseteq$ OBJECT.
Then operator
  DEMARK: OBJECT x OBJECT ——> OBJECT
is defined by

```
DEMARK($OB_1$, $OB_2$) :=
 Let {$ob_1$, ..., $ob_n$} = $OB_1$  in
 Let $f_i$ := (map $ob_i$), $i \in (n)$  in
 Let $F_i$ := {($ob'$, $f_i(ob')$) | $ob' \in OB_2 \wedge f_i(ob') \neq \bot$},
        $i \in (n)$  in
 Let $ob_i'$ := $\mu_0$($sel_1$: ($sel_1$ $ob_i$), $sel_2$: ($sel_2$ $ob_i$),
                map: (map $ob_i$) \ $F_i$,
                new: (new $ob_i$))
     where (case $ob_i \in$ Type_def_struct :
              $sel_1$:= type_id, $sel_2$ := type,
            case $ob_i \in$ Enr_def_struct :
              $sel_1$ := enr_id, $sel_2$ := enr),
      $i \in (n)$  in
 Let $Z_0$ := $OB_1$  in
 Let $Z_i$ := $Z_{i-1}$ \ {$ob_i$} $\cup$ {$ob_i'$}, $i \in (n)$  in
     $Z_n$
```

¤

Remark: DEMARK removes all occurrences of objects of its second argument set from the map component of the objects of its first argument set.

The next operator is helpfull to express the substitution induced by the map-component of an object.

Def. 3.7.3.-3 [S-MAP]
Let OBJECT := (Type_def_struct $\cup$ Enr_def_struct).
Let ob $\in$ OBJECT.
Then the operator
  S-MAP: OBJECT x Map ——> (Map x Map)
is defined by

```
S-MAP(ob, g) =
  if (map ob) = $\bot$ then $\bot$ else
   Let F = {($ob_i$, $ob_i'$) | $i \in (n)$} := (map ob), $n \in \mathbb{N}$ in
   Let G = {($op_i$, $g(op_i)$) | $i \in (m)$, $op_i \in$ OPS($ob_i$),
                                  $ob_i \in$ source(F),
                                  $g(op_i) \in$ OPS(target(F))}, $m \in \mathbb{N}$
                                                                 in
       (F, G)
```

¤

Remark: G $\subseteq$ (OPS(source(F)) x OPS (target(F))) together with F does not necessarily describe a signature morphism by this definition. This property has to be assured separately.

Def. 3.7.3.-4 [GENERATE]
Let OBJECT be defined as above.
Let OB $\subseteq$ OBJECT, $\bar{R}_u$(OB) hierarchical, g $\in$ Map such that Source(g) $\subseteq$ OPS(OB).

1) The operator

```
    GENERATE: $\mathcal{P}$(OBJECT) x Map —— $\mathcal{P}$(OBJECT)
is defined by
    GENERATE(OB, g) := let n := height(TOP(OB)) in
                       GENERATE1 (OB, g, n)
```

2) The operator

```
    GENERATE1: $\mathcal{P}$(OBJECT) x Map x N ——> $\mathcal{P}$(OBJECT)
is defined by
    GENERATE1(OB, g, n) := if n = 1 then OB else
     let {$on_1$, ..., $on_a$} := {ob| ob $\in$ OB and depth(ob) = n}
                                                                   in
     let $Z_0$ := OB in
     let $Z_{i+1}$ := (case (map $on_i$) = (new $on_i$) = $\perp$ : $Z_i$

                  case (map $on_i$) $\neq \perp$, (new $on_i$) = $\perp$ :
                   let $on_i$' := $\mu_0$($sel_1$: ($sel_1$ $on_i$),
                          $sel_2$: ($sel_2$ $on_i$)<S-MAP($on_i$, g)>,
                          map: {$\perp$}, new: {$\perp$})          in
                   let Z := DEMARK($Z_i$, {$on_i$}) in
                   let Z' := MARK(Z, {($on_i$, $on_i$')} \ $\bar{R}_u$($on_i$)
                                  $\cup$ $\bar{R}_u$($on_i$')) in
                         Z'

                  case (map $on_i$) = $\perp$, (new $on_i$) $\neq \perp$ :
                   let Z' := $Z_i$ \ {$on_i$} $\cup$ {(new $on_i$)} in
                         Z'

                  case (map $on_i$) $\neq \perp$, (new $on_i$) $\neq \perp$ :
                   let $on_i$' := $\mu_0$($sel_1$: ($sel_1$ (new $on_i$)),
                                 $sel_2$: ($sel_2$ (new $on_i$))
                                          <S-MAP($on_i$, g)>,
                                 map: {$\perp$}, new: {$\perp$})   in
                   let Z := DEMARK($Z_i$, {$on_i$}) in
                   let Z' := MARK(Z, {($on_i$, $on_i$')} \ $\bar{R}_u$($on_i$)
                                  $\cup$ $\bar{R}_u$($on_i$')) in
                         Z'
                  where (case $on_i$ $\in$ Type_def_struct:
                           $sel_1$ := type_id, $sel_2$ := type
                         case $on_i$ $\in$ Enr_def_struct :
                           $sel_1$ := enr_id, $sel_2$ := enr),

              i $\in$ (a)) in

          GENERATE1($Z_a$, g, n-1)
```

Remarks: a) New objects are constructed whenever (map on_i) $\neq$ $\perp$. This means that identifiers are substituted according to S-MAP(on_i, g). The incorporation of the new construct requires a re-marking of the hierarchy. The case (new on_i) $\neq$ $\perp$ is already covered by the MARK operator, and the incorporation of (new on_i) does not require a re-marking.

b) Application of GENERATE extends the object set in general. The hierarchical structure is preserved since subtrees are exchanged against subtrees and the TOP-element is left unchanged, i.e. (new TOP(OB)) = $\perp$.

c) The effect of GENERATE on ob $\in$ (OB $\setminus$ $\bar{U}(\bar{R}_u(ob))$) need not to be made explicit since the depth and height operator are based on $\bar{P}_R(ob)$ such that the 'intermediate' objects are included in $\{on_1, \ldots, on_a\}$.

3.7.4. Instantiate Type Definition

Before we state the semantic clauses for instantiate type definitions we will solve a technical problem. Many operators introduced up to now were based on the syntactic domains of the ModPascal semantics, and this was sufficient since no interactions to elements of semantic domains had to be expressed. The only and undangerous intersections of syntactical and semantical domains happend in the cases of Id, BOOL and INT.

The important point now is that syntactical operators as e.g. U (= the use relation operator) are not applicable, if their syntactic argument is exchanged by its semantics: in a state (ξ, σ) that involves the meaning Mm⟦mod⟧ of a module type object mod it is not possible to extract the set of used objects of mod in the current semantic domain structure. All information about them has been merged together by Mm, and currently the only way to get them is by looking at the syntactic object mod (where U(mod) is defined).

In the case of instantiate type definition semantics, much of the information gathered by syntactical operators should be accessable to the semantic operations (e.g. object hierarchy information, signature morphism properties, compatibility, substitution). The first step towards a connection of syntactical and semantical operators were done in 3.6.1.1. where additional components of semantic domains were introduced such that for example use relations could be modelled on the semantical level. But processing in this way would inevitably increase the complexity of the semantic domain structure, and finally each syntactic domain would have a semantic counterpart. A modelling of this kind is characterized by a very high degree of (unwanted) redundancy.

In the next definition a general mechanism is provided to overcome the deficiencies of level-separated operators. It links semantic objects to their syntactical definition uniquely and therefore represents an inverse meaning function. As a result, syntactic operators can be invoked on semantical objects by exchanging the arguments.

Def. 3.7.4.-1 [Retrieve]
Let $(\xi, \sigma) \in$ State and id $\in$ Id with $\sigma(\xi(\text{id})\downarrow 1) \neq \perp$ and $\xi(\text{id})\downarrow 2 \in$ AlgQual.
Let OBJECT := (Type_def_struct $\cup$ Enr_def_struct).
Then the operator
  Retrieve: Id ⟶ State ⟶ (OBJECT x State)
is defined by:

```
Retrieve(id)$\xi\sigma$ :=
      $\iota$ (ob, ($\xi_1$, $\sigma_1$)) $\in$ (OBJECT x State) .
        Let ($\xi_2$, $\sigma_2$) := M⟦ob⟧$\xi_1\sigma_1$  in
           $\xi_2 = \xi$ and $\sigma_2 = \sigma$ and (ob_id ob) = id
```

¤

Remarks: a) The selector ob_id represents type_id or enr_id depending on ob.

b) The injectivity of Mm/Me/Mt needs not to be assumed since the definition of retrieve is not constructive (ι-operator). Also, every ModPascal program is listed sequential, and the syntactic object that causes the next state transition is directly derivable.

c) From the definition, it follows:

```
$\forall$ id $\in$ Id, ob $\in$ OBJECT, ($\xi$, $\sigma$) $\in$ State .
if id = (ob_id ob) then
  if Retrieve(id)$\xi\sigma \neq \perp$ then
     Let ($\xi_1$, $\sigma_1$) := Retrieve(id)$\xi\sigma$)$\downarrow 2$  in
     Let ($\xi_2$, $\sigma_2$) := M⟦ob⟧$\xi_1\sigma_1$   in
         ob = (Retrieve(id)$\xi_2\sigma_2$)$\downarrow 1$
```

This constitutes a link between semantic and syntactic domains since identifiers are assumed unique for syntactic and semantic domains (see sec. 2.2.3.1.).

Notation: For I $\subseteq$ Id, (ξ, σ) $\in$ State .
RetOb(I)$\xi\sigma$:= {(Retrieve(id)$\xi\sigma$)$\downarrow 1$ | id $\in$ I}

For the syntactical operators now it is possible to use them in semantic clauses. For example, if I $\subseteq$ Id
  MARK(Retrieve(I)$\xi\sigma$, f)
is meaningfull and evaluates to an object set OB $\subseteq$ OBJECT. That could be subject to a transformation into the semantical level by
  M⟦SEQ(OB)⟧$\xi\sigma$.
(SEQ is applied to guarantee syntactical correctness).

Pictorially, we have

$$\begin{tikzcd}
I, (\xi, \sigma) \arrow[r, "\text{Retrieve}"] & OB \arrow[d, "\text{MARK/SEQ}"] \\
I', (\xi', \sigma') & OB' \arrow[l, "M"'] \\
\text{semantics} & \text{syntax}
\end{tikzcd}$$

After provision of this technical prerequisite, we turn back to the theme of this section.

The semantics of an instantiate type definition is computed by performance of the following steps.

a) Check, if the set of instantiation objects constitutes a hr signature morphism SM.
b) Check, if the object hierarchy spanned by the base object and the hierarchy spanned by the source objects of SM are compatible.
c) Mark in the base object hierarchy those objects that are subject to changes by the signature morphism.
d) Generate the new objects and incorporate them in the current environment except the new base object.
e) Return the modified base object as value for the instantiate type definition identifier.

Sem_17: Instantiate Type Definition

```
Mm⟦i: Instantiate_type⟧ρσ :=
  let bid := (base_type i),
      {$i_1$, ..., $i_n$} := (objectL i)   in
  let Bid := (Retrieve(bid)ρσ)↓1  in
  let {$I_1$, ..., $I_n$} := RetOb({$i_1$, ..., $i_n$})ρσ  in
  let I := $I_1$ + ... + $I_n$,  I = (f, g)  in
    if not (SM?(I)) = true then ⊥ else
    if not (Comp?(Bid, I)) then ⊥ else
      let $Bid_1$ := MARK($\bar{U}$($\bar{R}_u$(Bid)), f)  in
      let $Bid_2$ := GENERATE($Bid_1$, g),
            {$ob_1$, ..., $ob_m$} = $Bid_2$  in
      let objL := SEQ({$ob_1$, ..., $ob_m$})  in
      let ($ρ_1$, $σ_1$) := M⟦objL⟧ρσ  in
      let (A, ($ρ_2$, $σ_2$)) := M⟦TOP($Bid_2$)⟧$ρ_1σ_1$  in
          (A, ($ρ_2$, $σ_2$))
```

Remarks: a) The semantics of the base type and the used instantiation objects (both are elements of Id) are computed from the application state. By means of the Retrieve operator the associated syntactic objects are taken to perform the instantiation process (marking, object generation). The resulting object set is sequentialized and mapped to the appropriate semantic domain. The resulting state and the algebra of the TOP-element are passed.

b) All implicitly generated objects are installed. An appropriate naming procedure is assumed.

4. Precompilation

4.1. The Verification Environment

As pointed out in the introduction, ModPascal was developed as part of the ISDV-System that supports verifiability of software. This is realized by providing methods and tools for stepwise refinement from requirements specifications over applicative ASPIK structures to imperative ModPascal code and by methods and (semiautomatic) tools for verification of the refinement steps.

The final refinement step in this setting is the transition from algorithmic ASPIK specifications to ModPascal module type and enrichment definitions. One has to assure that, for example, a module 'does the same' as a specification. Since both objects are independantly specified/programmed this task is nontrivial, and without further confinements even unsolvable, because it is equivalent to the (undecidable) problem of showing that two arbitrary Turing machines behave identically. In filling the prosaic term 'does the same' with a formal content one has to solve the following tasks:

- Definition of semantical criterium that assures the correctness of the transition in a mathematical formalism.
- Specifying a method to (hopefully automatic) check in a concrete case if the correctness criterium is valid.

The ISDV-System provides a satisfactory solution fitting to the ModPascal/ASPIK environment. A detailed description can be found in [Olt 85], here we give a brief overview.

The applied correctness criterium is essentially based on the existence of semantic algebra domains that provide the meaning for modules and specifications, and on the notion of algebra homomorphism. If the semantic algebra of the specification is found to be a homomorphic image of the semantic algebra of the module then the transition is called correct; both objects do the same. This seems to be a weak condition, but in the ModPascal/ASPIK environment the existence of a homomorphism implies an isomorphism, and so the desired 'strong' criterium is achieved. Isomorphisms as correctness criterion is often used in abstract data type theory (e.g. correctness of extensions or implementations; see [EKP 78]).

The main problem is that the check of the validity is based on a set of equations that are most unlikely to be processed even by semi-automatic proof systems: they enclose ModPascal constructs as well as ASPIK terms and between, there are semantic functions as defined in section 3. Since this fact makes the correctness check of the ASPIK/ModPascal transition a human-bound task efforts were made to recognize and treat special situations in which mechanical support is possible. These situations are characterized by the structures occurring in the ModPascal object involved in the transition. If they do not leave an 'elementary' level, they can be transformed via symbolic evaluation in expression vectors that directly correspond to ASPIK terms. In that case the set of equations could be (semantically equivalent) expressed as pure ASPIK-equations, and then the check of validity would only have to deal with ASPIK structures. This does not imply the (semi-automatic) solvability of the equations, but it removes a degree of complexity from them.

In the ISDV-System there is a semi-automatic tool that generates for a given specification and a given module a set of equations that are possibly simplified to pure ASPIK equations. In the latter case the check of validity is initiated by passing them to one of the proof systems connected to the ISDV-System (e.g. MKRF [BES 81], RRLAB [Tho 84]).

The precompilation problem arises at the point when the original set of equations is checked for 'elementary' structure and the symbolic evaluation is performed. For both tasks software tools were available at the beginning of the ModPascal development, but they recognized only Standard Pascal. The solution to this problem was to precompile the ModPascal code into Standard Pascal code and then apply the desired tools. The necessary precondition was that the precompilation will not disturbe the special semantic structures associated with ModPascal constructs. This is non-trivial since semantical preservation is in general not a property of precompilation; only together with subsequent compilation with a 'verified' compiler this will hold.

In another view, the precompiler solution was preferable since the quantity of conformity of ModPascal and Standard Pascal greatly exceeds the quantity of differences, and Standard Pascal compilers are widely available. But again, because of the application of ModPascal in a system for verifiable software, one has to formally assure that precompilation is semantics preserving.

These issues justify the current section. We will specify the transformations performed for single constructs and show, that under the definitions of sections 2. and 3. isomorphic code is generated. A description of the more technical aspects may be found in [Eck 84] and [Sch 85]. The application in the generation of equation sets is documented in [Wei 85].

4.2. The Transformation

4.2.1. The Operator PRE

In this section we define an operator PRE that is applied to precompile ModPascal to Pascal. We refer to the abstract syntax of ModPascal of sec. 2.1.2.

Notations: $Constr_P$ denotes the domain of all correct Standard Pascal programs

$Constr_M$ denotes the domain of all correct ModPascal programs (= domain Program of 2.1.2.)

Since ModPascal extends Standard Pascal, it holds:
$Constr_M \subsetneq Constr_P$

With Sel_M, Sel_P we denote the set of possible selectors for objects of $Constr_M$ and $Constr_P$ resp. Again $Sel_P \subsetneq Sel_M$ holds. Then $Constr_P = \{o \in Constr_M \mid \neg(\exists\ s \in (Sel_M \setminus Sel_P)\ .\ (s\ o) \neq \bot)\}$

The explicit definitions of Sel_M, Sel_P are omitted here. They can be derived directly from 2.1.2. and an analogous abstract syntax for Standard Pascal.

In the following we view at $Constr_M$ and $Constr_P$ as the coalesced sum of all syntactic domains they are build upon (i.e. $Constr_M$ = Program + Prog_head + ID + Block + Lab + ...). This allows to define the operator PRE with a single arity, but makes it applicable to every substructure of the above domains.

<u>Def. 4.2.1.-1</u> [PRE]
Let $Constr_M$, $Constr_P$ be as above.
Then the syntactical operator
PRE: $Constr_M \longrightarrow Constr_P$
is defined by:

```
1) $\forall$ o $\in$ $Constr_P$ . PRE(o) := o

2) $\forall$ t $\in$ Type_def .
   if t $\in$ $Constr_P$ then [$\longrightarrow$ 1)] else
    if (type t) $\in$ Module_type then
       PRE(t) := $\mu_0$($s_1$: TypeL, $s_2$: Func_dclL)
     if (type t) $\in$ Instantiate_type then
        let {$u_1$, ..., $u_n$} := $\bar{U}(\bar{R}_u$(base_type (type t))) in
          PRE(t) := $\mu_0$($s_0$: {PRE(base_type (type t))},
                          $s_1$: {PRE($u_1$)}, ... ,
                          $s_n$: {PRE($u_n$)})

3) $\forall$ i $\in$ Inst_def . PRE(i) := $\bot$

4) $\forall$ p $\in$ Proc_stmt .
   if p $\in$ $Constr_P$ then [$\longrightarrow$ 1)]
      else PRE(p) := $\mu_0$($s_1$: Assg_stmtL)

5) $\forall$ v $\in$ Var .
   if v $\in$ $Constr_P$ then [$\longrightarrow$ 1)]
      else PRE(v) := $\mu_0$($s_1$: Var, $s_2$: Assg_stmtL)

6) $\forall$ o $\in$ Op_designator .
   if o $\in$ Proc_stmt then [$\longrightarrow$ 4)]
      else PRE(o) := $\mu_0$($s_1$: Simple_term)

7) $\forall$ e $\in$ Enrich_def .
   PRE(e) := $\mu_0$($s_1$: Func_dclL)
```

8) i) PRE possesses a homomorphism property for ObjL:
$\forall$ oL $\in$ ObjL .
PRE(μ_0(s_1: (first oL), s_2: (rest oL))) =
μ_0(s_1: PRE((first oL)), s_2: PRE((rest oL)))

ii) PRE possesses a homomorphism property for VarL:
$\forall$ vL $\in$ VarL .
PRE(μ_0(s_1: (first vL), s_2: (rest vL))) =
μ_0(s_1: PRE((first vL)), s_2: PRE((rest vL)))

Remarks: a) This definitions of PRE does not reflect any context-sensitive conditions. It is just the relation between the abstract domains associated to programming language constructs. An algorithmic definition is given below (see sec. 4.2.2).

b) Standard Pascal structures are mapped identically.

c) By PRE, modules are transformed in a sequence of type definitions and function declarations. For sequences of module definitions, the PRE image would violate the Pascal syntactic law that type definition part and subprogram declaration part have to be disjunct. Therefore requirement 8) i) is necessary. Also, PRE disparts variable declarations for module variables consisting of the variable indication and an initial assignment of value (see example 4.2.2.-3). Since variable declaration part and statement part are dispart in Pascal, PRE has to fullfill requirement 8) ii).

d) ModPascal procedure statements as well as operation designators may consist of several (subsequent) procedure and function calls ('extended dot notation', see sec. 3.2.4. of [OLt 84]). This fact is reflected by requirement 4 that converts procedure statements in statement lists. For function calls (requirement 6) simple terms are sufficient where the sequentiality is transformable to nesting depth.

e) Instantiation definitions could be modelled in Pascal, but only with great struggles. Since there are no existing Pascal compilers that are capable of handling them inside a (precompiled) instantiate type definition, they are disregarded here.

f) The treatment of instantiate type definitions involves implicitly the object generation algorithm described in sec. 3.7.3. Since the generated sequence of object definitions is stepwise transformable (according to requirement 8)), PRE(t) for t $\in$ Instantiate_type is derivable from the PRE values for each sequence element.

Definitions 4.2.1.-1 shows that the precompilation task consists mainly of syntactical manipulations. Only in the case of instantiate type definitions context sensitive conditions are required. The algorithms that realize PRE and the implementation are documented in [Eck 84]. We will illustrate PRE by examples.

4.2.2. Concrete Definition

In the following we apply PRE to concrete syntactic constructs since ambiguities are not possible. The examples illustrate the processing of the currently implemented ModPascal precompiler.

Example 4.2.2.-1: Module procedure call (4.2.1.-1, case 4)
Let "public procedure P(x_1: X_1, ..., x_n: X_n)" denote a public procedure of a module M, and V a Variable of type M.
Then, if

V.P(y_1, ..., y_n)

denotes a call of P, PRE("V.P(y_1, ..., y_n)") is defined as

V := M&P(V,y_1, ..., y_n)

In other words, module procedures become functions with extended functionality and new operation identifier (see example 4.2.2.-4), and procedure statements are transformend into assignments to the module variable.

In the case of extended dot notation (see sec. 3.2.4. of [Olt 84]) an appropriate sequence of assignment statements (possible with automatically generated intermediate variables for function occurrences in the operation designator) is produced. All assignment variables are simple (module) variables. For example, "V.OP_1(a, b).OP_2(c, d)" is equivalent to "V.OP_1(a, b); V.OP_2(c, d)". The PRE-image is "V := M&OP_1(V, a, b); V := M&OP_2(V, c, d)" (if OP_1, OP_2 are procedures). We skip the details of the transformation algorithm (see [Eck 84]). ¤

Example 4.2.2.-2: Module function call (4.2.1.-1, case 6)
Let "public function F(x_1: X_1, ..., x_n: X_n): Z" denote a public function of a module M, and V a Variable of type M.
Then, if

V.F(y_1, ..., y_n)

denotes a call of F, PRE("V.F(y_1, ..., y_n)") is defined as

M&F(V,y_1, ..., y_n)

In other words, module functions become functions with extended functionality and new operation identifier (see also example 4.2.2.-4). ¤

Example 4.2.2.-3: Initial operation call (4.2.1.-1, cases 5, 8)
Let "public initial I(x_1: X_1, ..., x_n: X_n)" denote an initial operation of a module M.
Then, if

V:M#I(y_1, ..., y_n)

denotes a call of I inside a variable declaration, PRE("V.M#I(y_1, ..., y_n)") is defined as

V: M; ...; V := M&I(y_1, ..., y_n)

That means, that variable declaration and initial value supply are disconnected and assembled according to Pascal syntax. The generated assignment constructs are inserted as the starting statements of the next nested statement part. The initial operation is renamed and converted into a function. ¤

Example 4.2.2.-4: Module type definitions (4.2.1.-1, cases 1, 8i)
Since the concept for transforming modules is very central and important, we give a more detailed example of a module type definition for QUEUE, where TASK denotes the kind of 'queued' objects, and PRIO is an operation of TASK:

```
type QUEUE = module
     use TASK;                                         (1)
     public procedure ENTER(T:TASK);                   (2)
            procedure LEAVE;
            function NEXT : TASK;
            function ISEMPTY : BOOLEAN;
            initial EMPTYQUEUE;
     local type T = array [1..100] of TASK;            (3)
           procedure SHIFT(AR:T, I:INTEGER);
           var A:T, PTR:INTEGER;
     localend;

     procedure ENTER;                                  (4)
               var i:INTEGER;
               begin i:=PTR;
                     if i=100 then QUEUE&ERRORPROCEDURE
                     else
                     while i>1 do
                        if T.PRIO>A[i].PRIO
                        then i:=i-1
                     SHIFT(A,i);
                     A[i]:=T;
                     PTR:=PTR+1;
               end;

     procedure LEAVE; (* omitted *)

     procedure SHIFT; (* omitted *)

     function NEXT; (* omitted *)

     function ISEMPTY; (* omitted *)

     initial EMPTYQUEUE; (* omitted *)

modend;
```

Then PRE("type QUEUE = ... modend") is defined as follows:

```
type QUEUE&T = array [1..100] of TASK;
type QUEUE = record A: QUEUE&T; PTR: INTEGER end;
function QUEUE&ENTER(M1: QUEUE, T: TASK): QUEUE; FORWARD;
function QUEUE&LEAVE(M1: QUEUE): QUEUE; FORWARD;
function QUEUE&NEXT(M1: QUEUE): TASK; FORWARD;
function QUEUE&ISEMPTY(M1: QUEUE): BOOLEAN; FORWARD;
function QUEUE&EMPTYQUEUE: QUEUE; FORWARD;
function QUEUE&SHIFT(M1: QUEUE; AR: T; I: INTEGER): QUEUE;
         FORWARD;
function QUEUE&ERRORPROCEDURE(M1: QUEUE): QUEUE;
       begin ... [exception handling] ... end;
function QUEUE&ERRORFUNCTION(M1: QUEUE): QUEUE;
       begin ... [exception handling] ... end;
function QUEUE&ENTER;
         var i: INTEGER;
         begin i := M1.PTR;
               if i = 100 then M1 :=
                 QUEUE&ERRORPROCEDURE(M1)
               else while i > 1 do if T.PRIO < M1.A[i]
                                   then i := i-1
               QUEUE&SHIFT(M1, M1.A, i);
               M1.A[i] := T;
               M1.PTR := M1.PTR + 1;
         end;
function QUEUE&LEAVE; (* omitted *)
function QUEUE&NEXT; (* omitted *)
function QUEUE&ISEMPTY; (* omitted *)
function QUEUE&EMPTYQUEUE; (* omitted *)
function QUEUE&SHIFT; (* omitted *)
```

Remarks: a) The module definition is translated into a sequence of (standard) type definitions and a sequence of function declarations. It is obvious, that in the case of several module type definitions, their PRE-image has to be rearranged, according to the Pascal syntax (requirement 8 of definition 4.2.1.-1).

b) In the sequence of type definitions there occurs every local type of the module, with a unique type identifier. Additional, the set of local variables is contracted in a record definition, named by the module identifier. This record represents the data on which the module operations are performed; it is called the module record.

c) Only functions occur. This is due to the fact, that there are limitations for the use of global variables in ModPascal module operations. The set of allowed global variables is restricted to the set of local variables of the module. Since these are now structured together in one record type, every operation of a module (procedure, function, initial) is convertable to show functional behaviour as follows:

- add the module record to the functionality of every operation
- substitute in the operations body every occurrence of a local variable by the associated record field variable.
- substitute in the operations body every occurrence of a module operation call by the associated PRE-image.

Then the operations will get a module record argument as actual parameter, modify it and return either this new object or a selected component, i.e. they can be viewed as (mathematical) functions. By this, one is able to simulate the behaviour of module operations very closely.

d) For initial operations, the treatment is slightly different. Their functionality remains unchanged, since they are intended to 'initialize' a new module incarnation, and therefore they should not be supplied with an actual parameter that is of just that structure. (Otherwise initial would not mean 'really' initial).

e) No difference is made between public and local operations since these distinctions make no sense in non-object-oriented environments.

f) Functions are firstly introduced by 'forward'-declarations. This models the mutual recursion of operations possible in a module type definition.

g) The function identifiers are made unique by prefixing with the associated module identifier.

h) There are special error operations (despite bewildering names, both are functions). They are associated to every module, and their PRE-image is a piece of Pascal code, that at call time prints values of the module record fields and branches to the program end. If more sophisticated error handling is needed, it has to be programmed by the user.

¤

Example 4.2.2.-5: Enrichment definition (4.2.1.-1, case 7)
We use the objects QUEUE and TASK introduced in the example before as basis of an explanatory enrichment definition.

```
enrichment E-QUEUE use QUEUE is
     add TASK
               procedure MERGE(T:TASK);
         QUEUE
               function LENGTH(I:INT):INT;
               procedure SWAP;
     addend;
     procedure MERGE;
               begin (* omitted *) end;
     function  LENGTH;
               begin (* omitted *) end;
```

```
      procedure SWAP;
                begin (* omitted *) end;
enrend;
```

Then PRE("enrichment E-QUEUE ... enrend") is defined as follows:

```
function TASK&MERGE(M1: TASK; T: TASK): TASK; FORWARD;
function TASK&LENGTH(M1: QUEUE; I: INT): INT; FORWARD;
function TASK&SWAP(M1: QUEUE): QUEUE; FORWARD;
function TASK&MERGE; (* omitted *)
function TASK&LENGTH (* omitted *) ;
function TASK&SWAP; (* omitted *)
```

Remarks: a) The enrichment defintion is translated into a sequence of function definitions. It is obvious that in the case of a sequence of several different object definitions their PRE-images have to be rearranged according to the Pascal syntax (see requirement 8, definition 4.2.1.-1).

b) Remarks c) - h) of example 4.2.2.-4 apply analogously.

¤

Example 4.2.2.-6: Instantiate type definition (4.2.1.-1, case 2)

As pointed out in remark e) of 4.2.2.-1, instantiate types effort a special treatment that involves some semantical algorithms.
Let I_1, ..., I_n denote instantiation definitions and B a (base) object.
Then, if

type B' = instantiate B by I_1, ..., I_n;

denotes an instantiate type definition, PRE("type B' = ...;") is defined as:

Let OB(B) := $\bar{U}(\bar{R}_u(B))$, and OB'(B) denote the set of (possibly) modified objects, if the signature morphism induced by I_1 + ... + I_n is applied (= the result of GENERATE) in
PRE(OB'(B))
(where B' is associated properly to the modified B)

In other words, the PRE-image of an instantiate type definition is the PRE-image of the modified hierarchy behind the base type.

Remark : We do not go into further details, since this kind of objects does not lie in the scope of applications that need precompilation as precondition (see sec. 4.1.).

¤

The last remark applies in full extend to instantiation definitions too. As a consequence, the currently implemented precompiler disregards them.

4.3. Semantical Preservation

The goal of this section is to show, that the application of the semantic function M to a ModPascal construct and to its PRE-image yields isomorphic results. More, the isomorphism will consist just of those renamings described in the previous section. The consequence of this semantical equivalence is that whenever Pascal instead of ModPascal is needed it may be exchanged by its PRE-image, and insights gained from precompiled code take directly over to the associated ModPascal constructs.

In the following we refer to the concrete definition of PRE as given in the examples of sec. 4.2.2. The renaming process that prefixes all items of a structure (module, enrichment) with the structure identifier is disregarded because this contributes only to the trivial isomorphism.

(a) Module procedure call

Let p $\in$ Proc_stmt, p $\notin$ Constr_P. Let p_1 := PRE(p). Let $(\xi, \sigma) \in$ State.

p $\in$ Simple_term:

i) $M\llbracket p\rrbracket\xi\sigma$ is the application of the store transformation of (op_id p) to (ξ, σ) (after evaluation of parameters). The state change $(\xi, \sigma) \longrightarrow M\llbracket p\rrbracket\xi\sigma$ is visible only in the value change of variables of GL(searchdef((op_id p))$\xi\sigma$), the local variables of the associated module (see Sem_3).

ii) $M\llbracket p_1\rrbracket\xi\sigma = M\llbracket \mu_0$(ass_var: (ass_var p_1), expr: (expr p_1))$\rrbracket\xi\sigma$. The converted procedure call now is the single term (expr p_1), a function invocation. According to 4.2.2., the result type is the associate module record type for searchdef(ob_id)$\xi\sigma$, i.e. the function call yields an incarnation of a record over the local variables (see Sem_4). Now, the assignment to (ass_var p_1) describes the state change on variables of the module record type.

Then we have

$$\text{i:} \quad (\xi, \sigma) \xrightarrow{M\llbracket p\rrbracket} (\xi_1, \sigma_1) \quad \text{(changes of local variable values)}$$

$$\text{ii:} \quad (\xi, \sigma) \xrightarrow{M\llbracket p_1\rrbracket} (\xi_2, \sigma_2) \quad \text{(changes of record variable values)}$$

The states (ξ_1, σ_1) and (ξ_2, σ_2) are isomorphic, since from i) and ii) it follows:

Let ob := searchdef(op_id)$\xi\sigma$ in
Let $\{lv_1, \ldots, lv_n\}$:= local variables of ob in

Let MT denote the module record type with fields lv_1, ..., lv_n in
Let m := (ass_var p_1), type(m) = MT in
$\sigma_1(\varrho_1(lv_i)\downarrow 1) = \sigma_2(\varrho_2(m))\downarrow i$, $i \in (n)$
and
$\forall$ id $\in$ Id . if id $\notin$ {m, lv_1, ..., lv_n} then
$\varrho_1(id) = \varrho_2(id) = \varrho(id)$

p ∈ Op_designator:

i) $M\llbracket p\rrbracket\varrho\sigma$ is the application of several store transformations (possibly with result). They are concatenated in Sem_3 to yield a result state on (var_id p). Since intermediate function calls do not contribute to the state change (no side-effects, see assumption 3.1.2.-2), we get a state transition from (ϱ, σ) to $M\llbracket p\rrbracket\varrho\sigma$ which is visible only on the value of those object variables that are referenced by the operation designator. Let V_1 denote the set of this variables. (Note that V_1 is only dynamically determinable.)

ii) P_1 denotes a sequence of assignment statements, where the assignment variables are either (var_id p) or (automatically generated) intermediate variables. Every sequence member involves a function call that returns a value of the module record type or a component thereof. Therefore each assignment can be treated in analogy to the case p ∈ Simple_term. The state change is then reflected by the change of values of all left-hand-side variables of the sequence. The set of this variables is V_2 := {(assg_var st) | $\hat{\exists}$ i ∈ [length(p_1)] . st = (first(rest^i p_1))}.

Then we have

i: $(\varrho, \sigma) \xrightarrow{M\llbracket p\rrbracket} (\varrho_1, \sigma_1)$ (changes of local variables of all $v \in V_1$)

ii: $(\varrho, \sigma) \xrightarrow{M\llbracket p_1\rrbracket} (\varrho_2, \sigma_2)$ (changes of (record) variable values of all $v \in V_2$)

The states (ϱ_1, σ_1) and (ϱ_2, σ_2) are isomorphic since from i) and ii) it follows:

1) $|V_1| = |V_2|$
2) $\forall v_1 \in V_1 . \hat{\exists} v_2 \in V_2 . \sigma_1(\varrho_1(v_1)\downarrow 1) = \sigma_2(\varrho_2(v_2)\downarrow 1)$
3) $\forall$ id $\in$ Id . id $\notin (V_1 \cup V_2) \Rightarrow \varrho(id) = \varrho_1(id) = \varrho_2(id)$

(b) Module function call

Let f ∈ Expr, f ∉ $Constr_P$. Let f_1 := PRE(f)
Let (ϱ, σ) ∈ State.

f ∈ Simple_term

i) $E\llbracket f\rrbracket\varrho\sigma$ is the application of the store transformation with result of (ob_id f) to (ϱ, σ). Assumption 3.1.2.-2 allows to disregard side-effects. Therefore no state changes occur. The result is a structure component of a module, and is delivered by $(E\llbracket f\rrbracket\varrho\sigma)\downarrow 2$. The computation is based on accesses to the local variables lv_1, ..., lv_n

of ob := searchdef(f)$\xi\sigma$.

ii) f_1 differs from f in the extended functionality and the substituted occurrences of local variable accesses. Let n denote the new formal parameter of the module record type on which the substitutions are defined. Again, no side-effects occur, and the state remains unchanged.

Since states are not affected, it remains to commpare the selected components. Because lv_i corresponds to $v{\downarrow}i$, we have

$$\sigma(\xi(lv_i){\downarrow}1) = \sigma(\xi(v){\downarrow}1){\downarrow}i,\ i \in (n).$$

From the fact that E is deterministic it follows that

$$(E[\![f]\!]\xi\sigma){\downarrow}2 = (E[\![f_1]\!]\xi\sigma){\downarrow}2$$

<u>f ∈ Op_designator</u>

i) $E[\![f]\!]\xi\sigma$ corresponds to the application of several store transformations with result, where intermediate pure store transformations may also occur (see Sem_4). The state change caused by the latter makes (ξ, σ) and $(E[\![f]\!]\xi\sigma){\downarrow}1$ uncomparable. The resulting component structure is dependent of the access to local variables of occurring modules and of the induced state change.

ii) The PRE-image of f is an expression. Since every operation was transformed into a function (except it was already), no state change occurs, and the dependance on module record values is constructable from the first argument of every function.

Then we have

i: $(\xi, \sigma) \xrightarrow{E[\![f]\!]} ((\xi_1, \sigma_1), r_1)$ (local variable accesses determine r_1)

ii: $(\xi, \sigma) \xrightarrow{E[\![f_1]\!]} ((\xi_2, \sigma_2), r_2)$ (module record variable accesses determine r_2)

The state modification $(\xi, \sigma) \longrightarrow (\xi_1, \sigma_1)$ can be skipped since the ModPascal semantics assumes side-effect freeness in the case of expression evaluation (see sec. 3.1.2.-2). Every change in the local variable values caused by f corresponds to change of a module record component caused by f_1 such that

$$\sigma_1(\xi_1(lv_i){\downarrow}1) = \sigma_2(\xi_2(m){\downarrow}1){\downarrow}i),\ i \in (n)$$

where m denotes the associated module record variable for lv_i and n the number of components.

But then r_1 and r_2 are selected in an isomorphic state, therefore $r_1 = r_2$.

<u>f ∈ S_Term</u>

This case can be reduced to one of the above by substitution of the signum by an appropriate function call.

(c) Initial operation call

Let $v \in Var$, $v \notin Constr_P$.
Let $v_1 := PRE(v)$, $(\xi, \sigma) \in State$.

i) $M[\![v]\!]\xi\sigma$ installs (idL v) in the environment and assigns an initial value that is a vector of the local variables after the invocation of (init v), which corresponds to a module procedure call (see Sem_5).

ii) v_1 only separates the tasks: first all variable declarations are performed. Then the initializations are elaborated as the primary assignment statements of the following statement part.

Then we have

i: $(\xi, \sigma) \xrightarrow{M[\![v]\!]} (\xi_1, \sigma_1)$ (simultaneous installation of variables and values)

ii: $(\xi, \sigma) \xrightarrow{M[\![v_1]\!]} (\xi_2, \sigma_2)$ (separate installation of variables and values)

Since this initialization of different variables is side-effect free, it follows that

$$(\xi_1, \sigma_1) = (\xi_2, \sigma_2)$$

(d) $M \in$ Module_type

As described in sec. 4.2., module type definitions are transformed into a sequence of function declarations and type definitions. This design is justified by the following facts:

- Types and operations are also discernible in module type definitions. The syntactic structure works as a bracket.
- The proliferation of local types does no harm since the restrictive use imposed by the context-sensitive ModPascal semantics is not liberated by the transformation process.
- The introduction of module records is just a reformulation of the vector of local variables.
- Module operations show function-like behaviour even if they are procedures. This is due to the confinement for every module operation definition that the set of global variables has an upper bound in the set of local variables. With the introduction of module records that enclose a slot for every local variable and the synchronous extension of operation functionalities by such a formal parameter type, all possible side-effects can be captured, and the precompiled operations yield values either of the
  - module record type, if they were procedures or initials, or of a
  - component type, if they were functions. In that case, the component type is identical to the result type of the ModPascal definition.

Especially the last point gives the semantical justification to precompile <u>all</u> module operations into side-effect free functions. Even if efficiency considerations will not coincide with this renunciation of procedures it is a preferable solution when looking at the application areas of ModPascal. Its use inside the ISDV-System (see sec. 1) in verification tasks with applicative languages (see sec. 4.1.) profits from functional modelling. Also, the elements of the domain Alg (= algebras) employ functions, a fact which allows the direct representation of ModPascal function declarations.

Let $t \in$ Type_def, $m :=$ (type t), $m \in$ Module_type.
Let $t_1 :=$ PRE(t), $(\varrho, \sigma) \in$ State.

i) $M[\![t]\!]\varrho\sigma$ is a state modification in which three main tasks are performed (see Sem_15):
   - new introduced items (public and local operations, types or variables) are installed in the environment
   - the semantic algebra associated to the module type definition is build up and installed.
   - the main program algebra is updated

   Since the second task is sufficient to describe the semantics of modules, it should be pointed out that the installation of all introduced items is only done for technical reasons because semantical clauses are easier to state. $(Mm[\![m]\!]\varrho\sigma)\downarrow 2$ is passed to M that installs the algebra as value of (type_id t) and updates MPA (main program algebra; see sec. 2.2.4.). The induced state change is visible in the enlarged environment $(M[\![t]\!]\varrho\sigma)\downarrow 1$ compared to ϱ (i.e. $\{id \mid id \in Id,\ \varrho(id) = \perp,\ (M[\![t]\!]\varrho\sigma)\downarrow 1)(id) \neq \perp\})$ and in the MPA extension (M-Val, M-F) (see Sem_15). The intermediate state in which the elaboration of the local type definitions is initiated is (ϱ_b, σ_b), while it is terminated in (ϱ_c, σ_c) (see Sem_15).

ii) $M[\![t_1]\!]\varrho\sigma$ is a sequence $c_1;\ c_2;\ \ldots;\ c_n$ of Standard Pascal constructs. Therefore the elaboration of every construct involves
   - installation of the new item (type, function) in the environment
   - updating of MPA.

   The resulting state differs from (ϱ, σ) in the enlarged environment and in the MPA extension caused by every construct of the sequence. The sequence starts with local type definitions $c_1;\ \ldots;\ c_i,\ i \in [n]$. Let $(\varrho_0, \sigma_0) := M[\![c_1;\ \ldots;\ c_i]\!]\varrho\sigma$, i.e. the state after elaboration of the local type definitions of t_1.

Then we have:

i: $(\varrho, \sigma) \xrightarrow{M[\![t]\!]} (\varrho_1, \sigma_1)$ (environment enlargement and MPA modification)

ii: $(\varrho, \sigma) \xrightarrow{M[\![t_1]\!]} (\varrho_2, \sigma_2)$ (environment enlargement and MPA modification)

Since

- for every local type definition in t there is a type definition in t_1
- for every operation definition o_def in t there is a function declaration t_1 such that $M[\![\text{o\_def}]\!]\varrho\sigma$ = $M[\![\text{f\_def}]\!]\varrho'\sigma'$,
- the cartesian product of the local variable types is taken as the TOI of t by M, which again is identical to the TOI of the module record that is generated by PRE in t_1
- MPA is enlarged by the same objects (in different sequences)

the states (ϱ_1, σ_1) and (ϱ_2, σ_2) are isomorphic:

```
Let I denote the set of new introduced item identifier in t
                                                          in
Let I' denote the PRE-image of I, $|I'| = |I|$ in
Let mid := $\iota$type_id t), mid $\in$ I in
Let $\{lv_1, \ldots, lv_n\}$ := denote the local variables of m in
Let Opid := $\{i \in I \mid \varrho_1(i)\downarrow 2 \in \{PROC, FUNC, INIT\}$ in
a) $\forall$ id $\in$ I, id $\neq$ mid .
   a1) $\varrho_1(id)\downarrow 2 \in ObQual \setminus \{VAR\}$ or
         $(\varrho_1(id)\downarrow 2 = VAR$ and $id \notin \{lv_1, \ldots, lv_n\})$
          $\Rightarrow \hat{\exists}\ id' \in Id' \,.\, \varrho_1(id) = \varrho_2(id')$
   a2) $\varrho_1(id)\downarrow 2 = VAR$ and $id \in \{lv_1, \ldots, lv_n\}$
          $\Rightarrow i \in (n) \,.\, TOI(\varrho_1(id)\downarrow 3) = (\varrho_2(mid)\downarrow 2)\downarrow i$

b) $\varrho_1(mid)\downarrow 3 = \varrho_2(mid)\downarrow 3$

c) For $(\varrho_b, \sigma_b)$, $(\varrho_c, \sigma_c)$, $(\varrho_0, \sigma_0)$ as above:
   c1) $\sigma_c(\varrho_c(main)\downarrow 1) \setminus \sigma_b(\varrho_b(main)\downarrow 1)$ =
              $\sigma_0(\varrho_0(main)\downarrow 1) \setminus \sigma(\varrho(main)\downarrow 1)$
   c2) $\sigma_1(\varrho_1(main)\downarrow 1) = \sigma_2(\varrho_2(main)\downarrow 1)$
```

This result is mainly based on the previous considerations on module procedure, function and initial calls. It states the semantical equivalence of t and t_1 by comparison of single items that cause state changes.

(e) e ∈ Enrich_def

Enrichments differ from modules in that they do not introduce new data but only new operations (for already given modules). This makes the same justifications applicable as for modules (see (d)) as the operation translation is concerned; therefore all enrichment operations are transformed to functions without changing semantics.

Let e ∈ Enrich_def, e_1 := PRE(e), (ϱ, σ) ∈ State.

i) $M[\![e]\!]\varrho\sigma$ is a state change that involves the installation of all operations of all addparts of the enrichment. A semantical algebra is computed and associated to the enrichment identifier. Also MPA is actualized by this data (see Sem_16).

ii) $M[\![e_1]\!]\varrho\sigma$ represents the elaboration of a sequence of function definitions. Each sequence element is installed in the state and enlarges the MPA (see Sem_2).

Then we have:

i: $(\varrho, \sigma) \xrightarrow{M[\![e]\!]} (\varrho_1, \sigma_1)$ (operation and enrichment object installation, MPA enlargement)

ii: $(\varrho, \sigma) \xrightarrow{M[\![e_1]\!]} (\varrho_2, \sigma_2)$ (function installation, MPA enlargement)

Since
- for every enrichment operation there exists exactly one precompiled operation definition (which is semantical equivalent)
- the enlargement of MPA is identical

the states (ϱ_1, σ_1) and (ϱ_2, σ_2) are isomorphic up to the fact that for eid := (enr_id e), it holds that $\varrho_1(\text{eid}) \neq \bot$ but $\varrho_2(\text{eid}) = \bot$ (see remark below):

```
  Let {op_1, ..., op_n} := (operationL e),
      oid_i := (op_id op_i), i ∈ (n) in
  Let eid := (enr_id e) in
  a) $\forall i \in (n) \,.\, \sigma_1(\varrho_1(oid_i)\downarrow 1) = \sigma_2(\varrho_2(oid_i)\downarrow 2)$
  b) $\sigma_1(\varrho_1(main)\downarrow 1) \setminus \sigma(\varrho(main)\downarrow 1) =$
         $\sigma_2(\varrho_2(main)\downarrow 1) \setminus \sigma(\varrho(main)\downarrow 1)$
```

<u>Remark</u>: The enrichment object of ModPascal is characterized as an 'only-use' object: it is not possible to generate incarnations of it via variable declarations. Enrichments may only occur in the use-clause of module type definitions where they extend the operation set of visible module objects. This fact makes it superfluous to install an enrichment-like variable in (ϱ_2, σ_2).

(f) i ∈ Instantiate_type

The precompilation of an instantiate type definition can be reduced to the precompilation of the generated object sets. Since the kind of these objects is either module or enrichment, cases (d) and (e) apply. But considering the verification context of sec. 4.1. instantiate type definitions play only a minor role, because the effect and translation of generic type generators are not treated. Therefore the details of semantics preservation of instantiate type definitions are skipped here.

(g) i ∈ Inst_def

Instantiation definition are similary characterized as instantiate type definitions (see (f)). Therefore they are disregarded here.

5. Summary

In this paper the procedural programming language ModPascal is supplied by a denotational semantics. To capture the meaning of the module-, enrichment- and instantiation concept, specific domains were introduced:

- a domain Alg consisting of algebras; elements of Alg are associated to all type definitions and to enrichment definitions
- a domain Map; elements of Map are associated to instantiation definitions that are used to realize the ModPascal parameterization concept.

This approach embeds main ideas of abstract data type theory:

Types are included in a more appropriate form than denoting them by sets: since algebras enclose data and operations, the module concept of ModPascal is supplied with a semantics derived from abstract data type theory. Also the important problem of admissability of operation calls on specific data is easily solved: if the operation is not among the operations that are contained in the algebra associated to a variable's type, then the call is not admissable. Or in other words: data may only be changed or accessed by explicitly defined operations (the 'module paradigm').

For the first time enrichments (of modules) are introduced in a procedural language. The ModPascal semantics also supplies them with an algebraic meaning by constructing an algebra that encloses the basing structure as well as the augmenting items.

Operations in ModPascal programs are mapped to algebra functions - independent of a procedure, function or initial declaration. This is possible since the effect of an operation invocation is modelled by considering the state change only on the global variables occurring in the operations body.

The parameterization concept involves signature morphisms (instantiation definitions) that represent an association of 'formal and actual' type and operation names. In the domain Map elements are mappings that fulfill the requirement of a signature morphism such that direct application in the generation of an instance of a (parameterized) object is possible. Together with the semantics of instantiate type definitions (a hierarchy of modules and enrichments), this is a formalization of the very flexible parameterization concept ModPascal provides.

The main purpose of the development of ModPascal was a provision of an adequate imperative language inside the ISDV scenario, that allows the definition and check of correctness criteria between applicative and procedural specification levels (see [Olt 85]). With the semantical framework defined here one is able to overcome the following (standard) problems:

- incompatibility of domains: objects of the applicative level are algebraic specifications that are associated to (initial) algebra semantics (see [ADJ 78], [BV 83]). Now objects of the procedural level (modules, enrichments, types in general) find their semantical value also in an algebraic domain. Therefore comparison of objects can be reduced to comparison of algebras.
- state-oriented semantics vs. functional evaluation: the effect of operation invocations were modelled as state changes. This made them incomparable with models usually employed for terms build of operations of algebraic specifications (function carriers). Now operations are associated to algebra operations of appropriate algebras, and it is possible to involve easily operations of different levels in the same context.
- correctness criterion: up to now most correctness criteria (e.g. total/partial correctness) were based on constructs involving predicate calculus formulas (Hoare-style verification). There, the assertion language was the main bottleneck that limited the expressive power of the concept. Now the correctness criteria can be based on well-known algebraic properties and features as algebra homomorphisms or isomorphisms. Then the relations between objects are of similar kind than it is frequently proposed in abstract data type theory.
- loss of expressive power: many applicative languages provide constructs that lack an appropriate counterpart in procedural languages. Therefore a transition might be problematic if these constructs occur. With the module, enrichment, instantiation and instantiation types of ModPascal this gap is narrowed.

Acknowledgements

I'd like to thank my collegues from the SEKI project at Kaiserslautern for stimulating and clarifying discussions and for not beeing discouraged in stormy times.
This research was supported by the Federal Ministry of Research and Technology under contract IT 8302363.

6. References

[ADA 80] The Programming Language ADA. Proposed standard document, US DoD. Springer, LNCS 106, 1981.

[ADJ 78] Goguen, J.A., Thatcher, J.W., Wagner, E.G.: An initial algebra approach to the specification, correctness, and implementation of abstract data types, in: Current Trends in Programming Methodology, Vol.4, Data Structuring (ed. R. Yeh)', Prentice-Hall, 1978, pp. 80-144.

[BES 81] Blaesius, K.H., Eisinger, N., Siekmann, J., Smolka, G., Herold, A., Walter, C.: The Markgraf Karl Refutation Procedure. Proc. 7th IJCAI, Vancouver, 1981.

[BV 83] Beierle, C., Voss, A.: Canonical Term Functors and Parameterization-by-use for the Specification of Abstract Data Types. University of Kaiserslautern, Memo SEKI-83-07, 1983.

[CIP 81] CIP Language Group: Report on a Wide-Spectrum Language for Program Specification and Development. Report TUM-I8104, Technical University of Munich, 1981.

[Don 77] Donahue, J.: On the Semantics of "Data Type". Technical Report TR 77-311, Cornell University, 1977.

[Eck 84] Eckl, G.: A Precompiler for ModPascal (in German). University of Kaiserslautern, Interner Bericht 121/84, 1984.

[EKP 78] Ehrig, H., Kreowski, H. J., Padawitz, P.: Stepwise Specification and Implementation of Abstract Data Types. Proceedings 5th ICALP, Springer LNCS, 62(1978), 205-226.

[Gor 79] Gordon, M.J.C.: The Denotational Description of Programming Languages. Springer, 1979.

[ISO 7185] International Organization for Standardization: Programming Languages - Pascal. ISODIS 7185, 1982-08-12.

[Olt 84] Olthoff, W.: ModPascal Report. University of Kaiserslautern, Memo SEKI-84-09, 1984.

[Olt 85] On a Connection of Applicative and Procedural Languages. Internal Report. University of Kaiserslautern, 1985.

[Rey 74] Reynolds, J.C.: Towards a Theory of Type Structure Colloquium on Programming, Paris, 1974.

[RL 85] Breiling, M., Eckl, G., Olthoff, W., Rainau, U., Schmitt, M., Weiss, P.: The RL-Handbook. Internal Report. University of Kaiserslautern, 1984.

[Sch 85] Schmitt, M.: Extension of the ModPascal Precompiler (in German). University of Kaiserslautern, 1985.

[SIEM 83] Pascal BS2000. User Guide (in German) SIEMENS AG, Muenchen, 2983.

[Sto 77] Stoy, J.: Denotational Semantics. M.I.T. Press, Cambridge (Mass.), 1977.

[Ten 76] Tennent, R. D.: The Denotational Semantics of Programming Languages. CACM, 19, p. 437-453, 1976.

[Tho 84] Thomas, C.: The Rewrite Rule Laboratory (in German). University of Kaiserslautern, SEKI-MEMO-84-01, 1984.

[Weg 72] Wegner, P.: The Vienna Definition Language. Computing Surveys, Vol.4, No. 1, March 1972.

[Wei 85] Weis, P.: The Compatibility Checker (in German). University of Kaiserslautern, 1985.