An algebric specification of a Pascal compiler

Joelle Despeyroux

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(3) 954 90 20

Rapports de Recherche

N° 209

AN ALGEBRAIC SPECIFICATION OF A PASCAL COMPILER

Joëlle DESPEYROUX

Mai 1983

AN ALGEBRAIC SPECIFICATION OF A PASCAL COMPILER

Joëlle DESPEYROUX

Résumé :

Il semble y avoir un intérêt croissant pour les types abstraits algébriques en tant qu'outils pour donner la sémantique d'un langage ou pour en spécifier la traduction. Ce papier présente une première expérience en vraie grandeur dans ce domaine. Nous présentons une sémantique algébrique de Pascal, puis une sémantique algébrique du langage de bas niveau P-Code. Enfin, une spécification d'un compilateur Pascal P-Code est donnée comme une implémentation du type abstrait source Pascal. Et nous terminons par un petit exemple de traduction.

Abstract :

There is a growing interest in abstract data types as a tool for specifying semantics of programming languages and for specifying translations. This paper present the first large scale experience in this area. An algebraic semantics of Pascal, using abstract data type is given. Then an algebraic semantics of the low-level language P-Code is given. Finally a specification of a compiler from Pascal to P-Code, by means of implementation of abstract data types, is given, with a small example of translation.

Mots clés : sémantique algébrique - types abstraits algébriques - méta-compilateur - une sémantique algébrique de Pascal - une sémantique algébrique du P-Code.

Keywords : algebraic semantics - algebraic abstract data type - meta-compiler - an algebraic semantics of Pascal - an algebraic semantics of P-Code.

AN ALGEBRAIC SPECIFICATION OF A PASCAL COMPILER

Table of contents

1. Introduction

This paper is an overview of the algebraic semantics of Pascal and the specification of a Pascal-P-Code compiler given in [Des 82]. We had two aims in this work. On one hand we proposed an algebraic semantics of Pascal. As a matter of fact abstract data types seem to be a good tool in semantics but had never been used to describe the semantics of a realistic language. On the other hand, we gave to the meta-compiler "Perluette" its first realistic example, by means of a second abstract data type describing a low-level language : P-Code, and a specification of the translation.

The theory used is based on algebraic abstract data types and has been developed in [Gau 80], [Gau 82]. This method has been designed to specify, produce and prove compilers. As a state of a program execution can be characterized by a set of axioms, it is viewed as an abstract data type. A state modification is a data type transformation, described by "substitutions" on the set of formulae which characterize it. Translations of languages correspond to representations of abstract data types.

After having, very briefly and unformally, recalling this method we shall present our two algebraic abstract data types - describing Pascal and P-Code - and our representation specification, ending with a small example of a translation.

2. Theorical foundations

As it has been said above, the method used is based on algebraic abstract data types. Very little about abstract data types is required here - for beginning investigation on that topic see, for instance, [GTW 78]. We shall only recall very briefly and unformally how to present an algebraic abstract data type and how to describe semantics of programming languages and to specify a compiler using this theory.

2.1. Semantics of programming language

The semantics of a language will be specified in two parts. The first one is an abstract data type describing the properties of the language. The second one is a set of semantic equations associating a semantic value to a program. In this context, the semantic value will be a - closed - term of the abstract data type. To illustrate this, let us pick up some features of the data type - describing Pascal - we shall present in section 3. An algebraic abstract data type is given by a signature - a set of sorts and operations on these sorts - together with axioms.

Example

```
Type  Int
Op
      ( ) → Int : Zero ;
      (Int) → Int : Succ ;
      (Int,Int) → Int : Add ;
      ...
Axioms
      Add(Zero,i) = i ;
      Add(Succ(i),j) = Succ(Add(i,j));
      ...
End   Int ;
Type  Var
Op
      (Var) → Int : Value-of (mod) ;
      ...
End   Var.
```

This example needs some comments.
The Add operation is defined on Zero and Succ, which are basic operations, called "constructors".
Value-of is said to be "modifiable". That means that its properties may be modified by the execution of a statement : they depend on the current state. This is not the case for example, of the Add operation. We shall describe later how to modify such operations.

To describe programming languages we shall often need operations - or even sorts - which, strictly speaking, don't belong to the language. For example the operation

(Var) → Type-name : Var-type

is a "hidden" one. It will not appear in the semantic value of a program. To define "equal" operation on a sort, and to specify errors, we shall need the hidden sort Logical (truth values).

Now, what about properties of the algebraic abstract data type, and what about errors ?

In the algebraic data type framework, one often expect a data type to be hierarchical, consistant and sufficiently complete. A data type is hierarchical if one can order sorts, defining each one only using those previously defined. Intuitively, a - hierarchical - abstract data type is consistant if it does not contain contradictory axioms, and it is sufficiently complete if the axioms are sufficient to describe all the properties which are observable from an underlying sort. Using data types for describing languages we shall ask for the two first properties but not for the last one, as we shall see in section 3.

To specify error we shall use [Gut 80] preconditions and failure cases :

pre (Value-of (v)) = Is-declared (v) ;
Equal (y, Int 'o') => failure (Div (x,y)).

We decide to use preconditions for static errors and failure cases for dynamic one.

A very useful notion to mention is that of union. We shall discuss this notion in 3.2.

Type Var = Union (Simple-var, Array-var, ...)
Op
(Var) → Value : Value-of.

Let us now introduce the sort corresponding to the statements. In our hierarchical construct it is the last sort defined, so it is the "type of interest".

Type Stmt :
Op
( ) → Stmt : Nop ;
(Stmt,Stmt) → Stmt : Seq ;
(Var,Int) → Stmt : Assign.

We previously said that the semantic value of a program is a-closed-term of sort Stmt. Among semantic equations we shall have, for instance :

$\mathcal{S}$(v:=e) = Assign($\mathcal{L}$(v), $\mathcal{V}$(e)) ;
$\mathcal{V}$(exp + term) = Add ($\mathcal{V}$(exp), $\mathcal{V}$(term)) ;
$\mathcal{L}$(id) = Var 'id'.

The intuitive meaning of a term of sort Stmt is a state modification. For instance suppose the formula

Value-of (Var 'v') = Int 'o'

is valid in state S_o . The meaning of

Assign (Var 'v' , Int '5')

is the state transformation of S_o into S_1 where

Value-of (Var 'v') = Int '5' is valid in S_1 and
Value-of (Var 'v') = Int 'o' is not.
and every other property that was valid in S_o is also valid in S_1.

So the properties of Value-of have changed : a state modification is a data type transformation. Intuitively, a state - of a data type T - is the set of formulae - verifying the axioms of T - valid at a certain point of the execution of the program. The initial state is the least one and the execution of a program will correspond to state modifications from this initial state.

Note that we shall use some axioms, for instance "Is-declared (v) = false", which we shall call "initial axioms". This particularly set of axioms is verified by the initial state but is not supposed to be verified by any other state of T. A state of T is verifying the set of "general" axioms of T.

To define a state modification we use two - predefined - tools :

Appl (apply) and Subst (substitution)

Appl : (Stmt, State) → State.

Examples :

Appl (Nop,S) = S ;

Appl (Seq(m1, m2),S) = Appl (m2, Appl (m1,S)) ;

For those operations which change the properties of a - modifiable - operation, we shall use the substitution :

Appl (Assign(v,e),S)

= Appl (Subst(Value-of(v),e),S).

The semantics of Subst is formaly defined to perform the appropriate state transformation. For formal definition of the concept of state and substitution see [Gau 82]. It would happen that one want to perform several substitutions at a time, not precising the order of the substitutions. In this case one can write

Subst (< f1(λ), f2(λ) >, < μ1, μ2 >)

which is formally defined to perform simultaneously f1(λ) ← μ1 and f2(λ) ← μ2.

Now notice that static and dynamic semantics are not separated here. To execute a program is to rewrite the term

Appl (< semantic-value >, < initial-state >)

which contain declaration elaborations and statement executions. As the first part of a compiler produced by the meta-compiler "Perluette" must be an implementation of the static semantics - of the language to be translated - it would be useful to give static semantics apart from dynamic ones.

For a suggestion to separate static and dynamic semantics, see [Des 82].

So the semantics of a programming language will be given by an algebraic abstract data type enriched by a data type Stmt and by substitutions.

2.2. Specifying a compiler. The system "Perluette"

To specify a compiler we shall give, for the source language, a source data type and semantic equations. To describe the translation, we shall give a representation function from the source data type to the target one. Finally the target language will be given by a target data type and a code generation function.

Notice that the meta-compiler Perluette only uses the syntactic part of the abstract data types (namely the operations), the axioms being used for proving the translation correct. A complete description of the system is given in [PDes 82].

So a translation is viewed as a representation of abstract data types. In practice, in the produced compiler, it corresponds to the rewriting of a term of the source data type into a term of the object data type.

For the translation, only representation of visible objects of source data type are necessary - hidden objects don't appear in the semantic value of a program -, while representation of hidden objects are necessary to prove the translation correct.

For proving the produced compiler correct, we shall prove that the axioms of the source data type are kept by the representation, i.e. properties of source language are kept. This proof is partially mechanized [Mad 83] using the theorem prover LCF.

It has been said that the translation is defined by a representation function. This function is an homomorphism from the source data type to the object one : we give the representation of each sort and operation of the source data type :

Type Int : value ;
repr (Add(i1,i2)) = add (repr (i1), repr (i2)).

A compiler uses a symbol table. So the repr function will use a "symbol table", and a meta-language for process it.

Type Var = address ;
repr (Var'i') = index (BP, OFF(i)).

where OFF is the offset of variable i and BP is the base-pointer (register).

Now, what about errors ? The preconditions will be taken into account in the semantic equations while the failure cases will be traited by the representation function. In the current version of Perluette, the user has to "implements" static contraints with a boolean "correct" attribute on the source language grammar, while writing another attribute which constructs the semantic value of the program. The second attribute is evaluated only if correct evaluates to true.

An operation which has a failure case is represented by an operation which has a corresponding failure case or by a sequence of operations including a test for that error with jump to a label "error", or with an error handler, depending on the object language.

It has been said that the representation function is a homomorphism of abstract data types. In fact, the experience we present in this paper had shown this is too strong a constraint. We shall discuss the problem and give a solution in section 5.

The method described here has been developed to specify and prove compiler. It is argued in this method that abstract data types - only presentation of abstract data type, without choosing particular models - are a good tool for that. Translation of languages is specified by a representation function of abstract data types and is implemented by a rewriting of a source term - semantic value of a program - to an object term.

3. The source language : Pascal

Several formal definitions of Pascal are available now so this section is not to be seen as yet another definition of Pascal but as a test of a method on a realistic language. In fact abstract data types seem to be a good tool for semantics but had never been used for a realistic language before we write this semantic of Pascal [Des 82].

This paper is just an introduction to this work, so we shall present only an overview of our semantics of Pascal, giving - almost - only the features of the abstract data type which are needed for the - very small-example of translation we shall give in section 5. We shall not give semantic equations as they are quite straightforward.

3.1. The abstract data type

The principal sorts of our abstract data types are

- Logical
- Type-name = Union (Simple-type, Set, Array, ...)
  Simple-type = Union (Scalar, Range)
  which corresponds to the Pascal "types"
- Value = Union (Simple-value, Set-value, Array-value ...)
  Simple-value = Union (Boolean, Integer, Char, Scalar-cste)
  which correspond to the values of the different Pascal objects.

A Scalar-cste is for example Scalar-cste 'blue' whose Pascal type is Scalar 'colour'. An Array value is the value of an array variable.

- Const corresponding to Pascal-constant
- Var = Union (Simple-var, Set-var, Array-var,...)
- Smt , the "type of interest", describing the instructions.

Now, let us give some operations and axioms of these sorts. The constructors of sort Type-name are the constants (Scalar 'integer',...). They are the only visible operations. To describe the notion of type compatibility be need the operation :

(Type-name, Type-name) → Logical : compatible-types

An array is described by the operations :

(Array) → Simple-type : Index-type (mod) ;
(Array) → Type-name : Component-type (mod) ;

Each value has a - Pascal-type :

(Value) → Type-name : value-type defined by
Value-type (Var-value(v)) = Var-type(v)

The constructors of type Simple-value are

(Scalar) → Simple-value : First ;
(Simple-value) → Simple-value : Succ.

We have of course the operations Add, Sub, Int-div ... on integers.

A variable has a value and a type and if it is an array variable we can access to each of its elements.

(Var) → Value : Value-of ;
(Var) → Type-name : Var-type ;
(Array-var, Simple-value) → Var : Index-var

These is two possible errors to the operation Index-var, a static one and a dynamic one :

```
Pre(Index-var(v,i))
      = Compatible-types (Index-type(Var-type(v)),
                          Value-type(i)) ;

Failure (Index-var(v,i))
      <= Is-out-of-range (i,Var-type(v)).
```

In sort Stmt,
among declarations we have

```
Op
(Var,Type-name) → Stmt : Var-decl ;
(Scalar, Scalar-cste-list) → Stmt : Scalar-decl ;
(Array, Simple-type, Type-name) → Stmt : Array-decl ;

Definitions

Appl(Var-decl(v,t),s)
      = Appl(Seq(Subst(Var-type(v),t),
            Subst(Is-declared(v),Logical 'true')),s) ;

Appl(Scalar-decl(s,ℓ),s)
      = Appl(Seq(Subst(< First(s), Last(s) >,
                       < Sv-car(ℓ), Sv-last(ℓ) >,
                 Succ-define(s,ℓ)),s) ;

Appl(Array-decl(t,n1,n2),s)
      = Appl(Subst(< Index-type(t), Component-type(t) >,
                   < n1,n2 >),s)
```

Note that the operation Scalar-decl uses the sort list. The semantics of Scalar-decl(s,ℓ) defines First(s) and Succ of each element of s - that is done by Succ-define.

Now we give the semantics of the assignment - when the variable concerned is a simple variable -, and of the repeat statement

```
Op
      (Simple-var,Simple-value) → Stmt : Assign
      (Stmt, Boolean) → Stmt : Repeat
```

Axioms

Appl(Assign(v,i),s)
= Appl(subst(Value-of(v),i),s)
b = Boolean 'true' ∈ Appl(m,s)
=> Appl(Repeat(m,b),s)
= Appl(m,s)
b = boolean 'false' ∈ Appl(m,s)
=> Appl(Repeat(m,b),s)
= Appl(Repeat(m,b),Appl(m,s))

Of course we have defined a General-assign

(Var,Value) → Stmt : General-assign

which use Assign or Array-assign ... depending on the type of the variable.

Now we have presented so far a Pascal without procedure and label. These are of course entirely defined in [Des 82]. Let's tell here very briefly and unformaly our ideas for describing them.

To define procedures we add two sorts : the first one corresponding to the identifiers, the second one to the notion of procedures calls :

Type Id = Union(Var-id,Type-id,Proc-id,...)
Op
(Id) → Proc-id : Father ; "static link"
Type Call
Op
( ) → Call : Current-call ;
(Call) → Call : Caller ; "dynamic link"
(Call) → Proc-id : Name

The constructors of sort variable will no longer be constants but the designation of an identifier in a procedure call :

(Id,Call) → Var : Des

Then a term of type Var can be, for example :

Des(id,call),
Index-var(Des(id,call),Value '2') ...

The possibilities of jump in Pascal are very large. But just because one can jump forward and backword, we need a way to "memorise" the program itself. In fact we need a tool something like the concept of continuation in denotational semantics. In our framework, we add a parameter to the operation Appl

Appl(m,p,s)

will then have the intuitive meaning : execute the stmt m with program p and state s. We need the operations :

(Stmt,Label) → Logical : Exit ;
(Label) → Stmt : go-to :
(Label, Stmt) → Stmt : Labeled, Cont.

Exit(s,ℓ) is true if there is a jump to ℓ going effectively out of s. The sequences will have the same meaning as before if there is no exit from the first stmt. Intuitively Cont(ℓ,m) is - axiomaticaly defined to be - the part of the program m starting at the label ℓ. And so jump can be defined as follow :

Appl(Seq(go-to(e),m),p,s)
= Appl(Cont(e,p),p,s).

3.2. Some remarks

Auxilliary sorts, parameterized types.

To describe a real programming language we need numerous hidden operations (value-of,...) which correspond to some notions in the language. We also need numerous - partly - visible sorts which don't correspond to any notion in the language : these are auxilliary types. In our case we need different list types to describe the structure of a Scalar type, of a Record ... .

For these list types we really need parameterized types. Parameter passing have been largely studied [Ehr 78], [ADJ 80], but there still remains some problems with implementation of parameterized data types [EK 82], [SW 82].

The properties of the abstract data type

Our abstract data type is hierarchical. But its hierarchical presentation is not natural. For instance certain operations, expressing properties of Scalar or Record and using Lists, cannot be defined in Type-name sort but must be defined in a List sort.
Anyway, from a theoretical point of view, the abstract data type must be hierarchical, and it is so.

We must notice that we have written this data type in a very modular way. Introducing a new concept in the language never leading to come back to the types previously defined.

Our data type is expected to be consistent. Say, if it is not, the error could be detected by a mechanized tool "a la" Knuth-Bendix.

We have not the sufficiently completeness. Even without the Stmt sort, our data type is not sufficiently complete. This seems a too strong propertie for real programming languages. Pascal is not complete for several reason. For instance the order of characters is implementation dependent. The semantic of "Dispose" depends on the system. The first incompleteness can be eliminated while specifying the compiler, however the second one cannot.

Another kind of incompleteness comes from the fact that variables are not initialised. There are no axioms on the operation Value-of. In fact there are no axioms - except possibly initial ones - on a modifiable operation. The execution of a program makes the type "less undefined".

Union of types

We have said that we need the notion of union of types. It appears in this experiment that we need it for domains of operations and also for codomains :

```
Type Var = Union (Simple-var, Array-var)
Op
     (Var) → Value : Value-of
     (Array-var,Simple-value) → Var : Index-var
```

An union in a domain of an operation leads to the problem of coercions [Gog 78]. An union in a codomain of an operation is more difficult to handle. The problem is that signatures are not sufficient any more : is Index-var(v,x) belonging to the sort Simple-var, or Array-var ... ? The question is "what are the internal operations of sorts Simple-var, Array-var ... ? The problem of union of types is still an open problem in the algebraic framework.

4. The object language P-Code

Our aim here is to give a semantics of a low level language. Doing this we shall give the second part of the specification of our compiler. We shall not give the code generation function as it is quite straightforward, except for optimisations.

P-Code is an intermediate language for Pascal. We must note that P-Code is not strictly speaking a language, because its instructions have a "meaning" only with respect to a given P-machine (a stack-machine), i.e. a certain memory organisation, a system and a P-Code "language". There is no standard for P-Code. We have chosen here the UCSD P-Code, slightly modified to be clearer and cleaner. Manuals of P-Code speak about integers, sets and procedures but P-Code only knows about words and addresses. In fact, apart from some points discussed below, we can describe P-Code as if it was a real language, keeping in mind that nothing is done to verify that a P-Code program really corresponds to a certain Pascal program and that the memory organisation is "correct".

4.1 The abstract data type

The main sorts of our abstract data type will be

word = union (value, address, label) and stmt.

In the sort word we shall find an operation which gives the memory contents of an address :

(address) → word : cont

For the addresses we shall have pointers and an operation of indexation, and for the values : min-value, succ, add, ... :

( ) → address : BP "Base-Pointer", SP "Stack-Pointer" ;
(address,value) → address : index.

As the Stmt sort mainly consists of operations on the execution stack referenced by SP, we shall define hidden operations push, pop and store :

```
appl(push(w),s) = appl(seq(subst(SP,ad-pred(SP)),
                           subst(cont(SP),w)),S) ;

appl(pop,s) = appl(subst(SP,ad-succ(SP)),s) ;

appl(store(a),s) = appl(seq(subst(cont(a),cont(SP)),
                            pop),s)
```

where ad-pred and ad-succ are the operations predecessor and successor on the addresses. A lot of operations will use these operations :

```
appl(ℓdci(v),s) = appl(push(v),s) ;

appl(sto,s) = appl(seq(store(cont(ad-succ(SP))),
                       pop),s) ;

appl(ℓao(v),s) = appl(push(index(BP,opp(v))),s).
```

Some other ones use the hidden operations on values :

```
appl(adi,s) = appl(seq(subst(cont(ad-succ(SP)),
                             add(cont(ad-succ(SP)),
                                 cont(SP))),
                       pop),s).
```

The possible errors here are essentially overflow :

```
not(is-word(add(v1,v2))) => failure (add(v1,v2))
```

where is-word define the acceptable range of a value.

4.2. Some remarks

Differences between both data types

The abstract data type used to define Pascal and P-Code look very different. Pascal allows using of complex expressions and has few statements. So the main part of the abstract data type which describes it describes data structures. The sort Smt has few operations. The data type has numerous hidden operations. In the contrary P-Code is almost entirely composed of instructions. So the main part of the abstract data type consists of - visible - stmts. The value sort is almost completely hidden.

Optional failure

To specify an error the system can handle, we do not use the "ordinary" failure case presented in section 2.1. We use the optional failure", written "failure (f(x)) => ℓ", meaning that if ℓ is valid then the error may happen. This allows, for instance, the implementation to have a garbage collector. The problem is that we can't forbid a garbage collector nor specify a particular one since we want to define a language, not a language with a particular system.

Implementation dependency

We have succeeded in describing P-Code without considering any particular implementation except for one point : the operations on word blocks "implementing Pascal's sets". We have described these operations using operations on integers but we cannot avoid supposing a particular implementation of these integers. The problem is that we need to define such operations as bit-test or fix-this-bit-to-one, as P-Code only knows about words.

Initial state of the memory

P-Code is not, strictly speaking, a language : the state of the memory, just before the execution a P-Code program, depends on that program. The memory parts concerned are those reserved for constants, procedures attributs and local variables of the main programs of ... the program of high-level language whose P-Code program is the translation !

A solution, for the abstract data type describing P-Code, is to consider that the initial state of the abstract data type will contain properties defined by the axioms - as usual - and some properties depending on a particular program.

In the case of a translation these formulae will be constructed by the representation as they depend on the source program.

5. The representation

We shall present here a specification of a translation from Pascal to P-Code. We must note that, in [Des 82] we have entirely described Pascal but we have only translated a subset of Pascal not containing procedures and labels. We have only represented visible objects, as hidden ones are not necessary for the translation itself but for the correctness proof which is out of the scope of this work but is discussed in [Mad 83].

It has been said in section 2.2 that a representation fonction may be not sufficient to specify a translation. Suppose - as it is the case here - that we have a stack language and we want to translate an Assign. We would like to write

```
repr (Assign(v,i))
     = seq (push (repr(i)),
            push (repr(v)),
            store).
```

Now what is repr(i) ? "i" may be Add (x,y), ub(x,y), ...
Suppose our object language has only stmt - as it is the case here and as it is generally the case in low level languages. It becomes impossible to use only the homomorphism "repr".

But anyway we don't want to represent an addition by a stmt because we don't want to make program proofs. As we said before the method presented here allows proof of the representation by reasoning on the properties of the operations (Add,...). So if the object data type is not rich enough, we must enrich it by an "auxilliary type" (containing add ...).

A term of the abstract data type describing Pascal is rewritten into a term of the P-Code type enriched by the Auxilliary type, with the representation function. Then this term will be rewritten in a term containing only visible objects of P-Code type, using the axioms of Auxilliary type, oriented to form a rewriting system. Schematicaly,

a source term will contain declarations and assignments, an intermediate term will contain the operations push, add..., while the object term - very closed to the code - will only contain operations of P-Code such as ℓoa, ℓdci, adi ... .

Now we shall give some features of the representation function and of the auxilliary type, and we shall end with an example of a - very small - program, giving all the different terms involved.

5.1. Representation function

Some objects of source type don't need to be represented. For example the sort Name-type is only need for static semantics and has nothing to do with the translation.

A source value will be represented by an object value or by an address :

```
Type Simple-value : value ;
Type Array-value : address ;
repr (Simple-value 'a') = value 'CONVERT(a)'
repr (Add(i1,i2)) = add(repr(i1), repr(i2))
```

where CONVERT(a) is, for example, the hexadecimal value of a, or the range of a in its Scalar type, depending on a. CONVERT is a function of a meta-language (1), which in Perluette is lisp.

A source variable will be represented by an address,

```
repr(Var 'I') = index (BP, value 'OFF (I)')
```

where OFF (I) is the associated offset of the variable I, put in the symbol table when representing the declaration of I, as we shall see later. The representation of Value-of (v) will simply be the memory contents of v :

(1) Functions of the meta-language, generally constructs or access on the symbol table, are written in upper-case.

repr (Value-of (v)) = cont (repr (v))

For representing an access to a structured variable, we shall use an operation index-var belonging to the auxilliary type :

repr (Index-var (v,i))
= index-var (repr(v), repr(i), SIZE(v), LB(v), UB(v))

where the operations SIZE, LB and UB will be used - in auxilliary type - for generating tests on value i.

As expected a source Stmt is represented by an object stmt. A declaration is represented by the stmt nop - without effect on the state - but when declaring a variable we have to allocate space for it : this is done by ALLOC, which create an entry in the symbol table for this variable and assigns it an address in the stack

repr(var-decl(v,n))
= (## ALLOC(v) ##)
nop ;

In the same way, the representation of a scalar or an array, ... , declaration will be nop and will consist in filling the symbol table in.

The representation of a simple assignment will be

repr(Assign(v,i))
= seq (push (repr(v)) ;
push (repr(i)) ;
< possibly some tests >,
sto).

For representing repeat instruction, we shall need an operation LABEL-GEN which generates an label :

```
repr (Repeat (m,b))
     = (## e := LABEL-GEN ##)
        seq(labelled(e,repr(m)),
            push (repr(b)),
            fjp(e)).
```

where labelled (e,m) is the stmt m labelled by e and fjp is a "false jump".

5.2. Auxilliary type

The main part of the auxilliary type consist of the definition of the operation push according to the kind of its argument :

```
push(value 'v') = ℓdci(v) ;
push(index(BP,value 'v')) = ℓao(v) ;
push(cont(a)) = seq(push(a),sindo) ;
push(add(v1,v2)) = seq(push(v1), push(v2), adi) ;
push(index-var(a,v,s,ℓb,ub))
     = seq(push(a),push(v),
            ℓdci(ℓb), ℓdci(ub), chk,
            ℓdci(ℓb), sbi, ixa(s)).
```

where the semantics of chk is pop twice while its failure case corresponds to an access out of the range of the index-type of v. The operation ixa performs the indexed access, taking its arguments on the stack.

Now, we did not deal with the problem of initial memory state. This initialisation cannot be made in the P-Code language. But we do not want to deal here with the interactions between languages and systems. So we decided to have an operation Execute in the source sort "stmt", represented by an operation execute.

But the semantics of execute is only given by comments. To execute a statement m mean to fill up the memory with tables constructed while executing the representation, and then to call the procedure m.

Now, a last point to mention concerns the way we treat those Pascal incompletenesses which are implementation dependent. Of course we cannot represent an incomplete Pascal. So we complete it by addition of axioms expressing those dependencies. So we are representing a particular Pascal, distinguished from others by a particular set of axioms.

5.3. An example

Now we are ready for a complete - very small - example of translation. We give here a program P, its semantic value t1 - source term -, the intermediate term t2 and the object term t3, which differs from the code only for comma and parenthesis - and for optimisations.

```
P = Program pascal
     Type Colour = (blue,red,black)
          Ar = array [colour] of integers
     Var i : integer ; a : ar ;
     begin i := 2 ; a [blue] := i+3 ; end .
```

$t_1 = \varphi(p)$ = Seq (Scalar-decl(Scalar 'colour', [blue,red,black]), (1)
Array-decl (Array 'ar',Scalar 'colour',Scalar 'integer'
Var-decl (Var 'i', Sclalar 'integer'),
Var-decl (Var 'a', Array 'ar'),
Assign (Var 'i', Integer '2'),
Assign (Index-var(Var 'a',Scalar-cste 'blue'),
Add(Value-of (Var 'i'), Integer '3'))).

(1) [blue,red] stands for Cons(Scalar-cste'blue', Cons(Scalar-cste'red',nil)).

t_2 = Repr (t_1)

```
    = seq (push(repr(Var 'i')), push(repr(Integer '2')),Sto      i=2 ;
           push(repr(Index-var(Var 'a',Scalar-cste 'blue')),     a[blue]
           push(repr(Add(Value-of(Var 'i'),Integer '3')),Sto)    :=i+3

    = seq(push(< i >), push(value '2'),sto        (1)
          push(index-var(< a >),value '0',...),
          push(add(cont(< i >),value '3')),sto)
```

$t_2 \overset{aux}{\rightarrow} t_3$

```
t3 = seq(ℓao([i]),ℓdci(2),sto,                                    i:=2
         ℓao([a]),ℓdci(o),                                        a[blue]
         ℓdci(o),ℓdci(2),chk,ixa(1)
         ℓao([i]),sindo,ℓdci(3),adi,sto.                          :=i+3
```

6. Conclusions

This work was a first large scale experiment with algebraic abstract data types used for giving the semantics of a language on one hand, for specifying a compiler, with the help of the system Perluette here, on the other hand.

Algebraic abstract data types seem to be a good tool for defining semantics of programming languages, because of their great modularity and - of course - abstractness. In this experiment we lacked a good specification language, possessing parameterised types and the union of types. It would also be useful to have a types library containing Logical, Integers, ... Now it is hoped that this experiment could be useful for describing other languages like Ada for instance.

Presentations and representations of abstract data types seem to be a convenient tool for specifying translators. A translation is specified as an implementation of abstract data types and implemented as a term rewriting. The translation is correct if it keeps the properties

(1) < i > stands for index(BP,OFF(i)).

of the source language - the axioms of the abstract data type. Now we must notice two features which are not in the theory yet. First of all we did not formalise the concept of "symbol table" used by the representation function. Then we said that homomorphisms of signature are not sufficient for describing the translation and we enriched the object data type by an auxilliary data type. We did not ask for the sufficient completeness of this enrichissment. In fact we asked for the sufficient completeness of a subpart of it : the part corresponding to the set of "possible intermediate terms" produced by the translation. This is not a usual requirement - and we did not present it formally here - but it is what we need.

So this experiment is a "proof of feasability" of abstract data types and points out several useful and interesting areas to investigate in.

Acknowledgements

Thanks go to Marie Claude Gaudel for having designed the theory without which this work would not have been possible, to Martin Jourdan for patiently reading the manuscript and to Claudine Decalf for typing.

References

[ADJ 80] H. Ehrig, H.J. Kreowski, J.W. Thatcher, E.G. Wagner, J.B. Wright
Parameterized data types in algebraic specifications languages, ICALP'80 - LNCS 88, pp 157-168.

[Des 82] J. Despeyroux
Une sémantique algébrique de Pascal et application à la spécification d'un compilateur Pascal-P-Code, Thèse, Octobre 1982.

[PDes 82] P. Deschamp
Perluette : a compilers producing system using abstract data types, International Symposium on Programming, Turin, April 82.

[Ehr 78] H.D. Ehrich
On the theory of specification, implementation and parameterization of abstract data types, Research Report, Dortmund 1978.

[EK 82] H. Ehrig, H.J. Kreowski
Parameter passing commutes with implementation of parameterized data types, ICALP'82, Aarhus - LNCS 144.

[Gau 80] M.C. Gaudel
Génération et preuve de compilateurs basées sur une sémantique formelle des langages de programmation, Thèse d'Etat 1980.

[Gau 82] M.C. Gaudel
Correctness proof of programming language translations, IFIP TC-2, Garwish-parktechnischen, June 1982.

[Gog 78] J.A. Goguen
Order Sorted Algebras : Exceptions and Error Sorts, Coercions and Overloaded Operators, UCLA - Rapport 14.

[GTW 78] J.A. Goguen, J.W. Thatcher, E.G. Wagner
Abstract data types as initial algebras and the correctness of data representation, Current trends in programming methodology, 1978, pp. 80-149.

[Gut 80] J.V. Guttag
Notes on type abstraction (2nd version), IEEE transaction on Software Engineering, 1980.

[Mad 83] E. Madelaine
Système d'aide à la preuve de compilateurs, Thèse à paraitre.

[SW 82] D. Sannella, M. Wirsing
Implementation of parameterized specifications, ICALP'82, Aarhus - LNCS 144.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique