Specification of compilers as abstract data type representations

M.C. GAUDEL (INRIA, FRANCE)

Summary

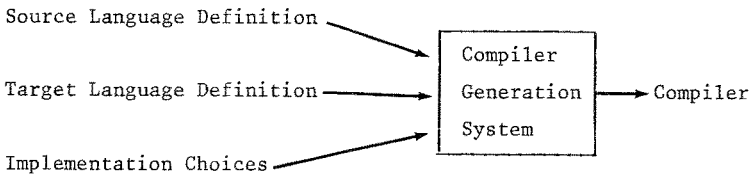
This paper presents a method for specifying and proving compilers. This method is based on the algebraic data types ideas. The main points are :
- to each language is associated an algebraic abstract data type,
- the semantic value of a program is given as a term of this data type,
- the translation of the semantic values of source programs into semantic values of target programs is specified and proved as the representation of an algebraic data type by another one.

A compiler generator, PERLUETTE, which accepts such specifications as input is described. The proof technic is discussed.

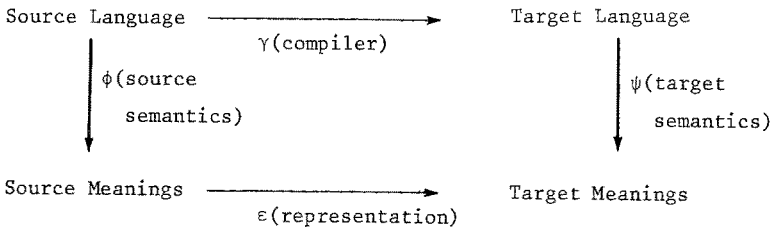
I - INTRODUCTION

Several experiments have been or are currently performed in order to use formal semantic specifications in a compiler generator. Although automatic production of parsers from a B.N.F.-like specification of a grammar is now widely known and used (see [1, 2] among many others), it still remains that the development of the other parts of a compiler is made, most of the time, in an 'ad hoc' manner. Tools like Knuthian attributes (W-grammars, Affix-grammars) are useful to specify compilers but the generation of all the parts of a compiler is possible only if there exists a formal and well-suited way to describe languages semantics. Of course, this description must be directly acceptable by the compiler generation system.

Among the works in this area, one of the most interesting is the SIS system developped by P. Mosses [3, 4]. This system takes as input a denotational definition of the Source Language semantics. The resulting translator compiles every source program into an expression of the intermediate language LAMBDA. This expression can be then evaluated by the LAMBDA interpreter.
Our approach [5, 6, 7, 8] is a bit different since our goal is to produce compilers which generate any given target code. Accordingly, a definition of both the Source and Target languages must be given to the system as well as the implementation choices :



In his paper 'advice on structuring compilers and proving them correct' [9] F.L. Morris stood by a similar idea : as he pointed out, it is necessary to specify source and target semantics ; moreover this approach makes it possible to get a correctness proof of the specified compiler, proving that the following diagram commutes.



Depending on the theoretical framework used, meanings of programs can be of various sort : continuous functions, predicate transformers, algebraic values, ... However we think that a well suited formalism to specify and prove representations is given by algebraic data types [10, 11, 12]. Thus, the semantic value of a program is considered to be a term of an algebraic data type and the bottom line (ε) of the diagram above is nothing more than the representation of a 'Source data type' by a 'Target data type'. This representation must be proved correct in the sense of [10] or [11] : the commutativity of the diagram is, obviously, not sufficient since it is true in the trivial case where there is only one value in the Target semantic domain !

We are currently developping a system, PERLUETTE[1], which generates a compiler from such specifications. The system is presented in part II of the paper. In part III an example of compiler specification is given. In part IV the proof method is outlined.

II - GENERAL PRESENTATION OF THE SYSTEM

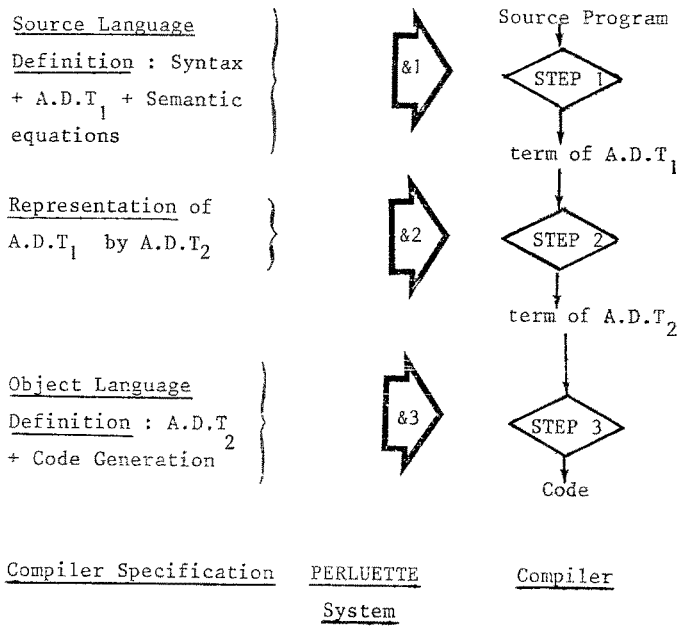
PERLUETTE takes into account only the 'syntatic' part of the abstract data types associated with Source and Target Languages. It is a compiler generator, not a prover. The axioms (the 'semantic' part) of the data types are needed for the correctness proofs which, at this time, are done by hand. It is definitely possible (and highly desirable) to mechanize these proofs using or modifying an existing system such as LCF [13], AFFIRM [14,15] or PROLOG.

The first part of a source language definition is the presentation of the algebraic data type associated to this language. This data type describes in a formal way the properties of the operations (using the more general meaning of this word) of the language. The semantic value of a program is a composition of some of these operations, i.e a constant term of the data type. Semantic equations specify the semantic value of a phrase of the programming language as a composition of the semantic values of the components of this phrase.

---

[1] Production Elegante et RelLlement aUtomatiquE de TraducTEurs - 'Perluette' is the french name of the character '&'.

These equations define, in a syntax directed way, a semantic function which has programs as domain and constant terms as co-domain.
The general scheme of the system is given below :



The resulting translators work in three steps : the first one produces an intermediate text, which is a term of the Source data type ; this term is rewritten by the second step as a term of the Target data type ; the last step performs the code generation starting from this second term.
In the current version of the system, the semantic equations of the source language are programmed using synthesized attributes [16, 17]. The language used to write these attributes is LISP. Attributes evaluation (and parsing) is performed by the DELTA system previously developped by B. Lorho [18]. The two parts of DELTA, the constructor and the evaluator have been respectively introduced in the first part of the generator (&1) and the first step of the translators (STEP1). The result of the first step is a LISP form, since the attributes

are specified in LISP. The STEP2 of the translators must evaluate this form using a LISP subsystem which is the result of &2. This subsystem is a set of LISP functions which are built from the representation specification : every operation name of the Source data type which can occur in the first term is considered, at this step, to be the name of a LISP function whose result is the translation of this operation into a composition of Target data type operations.

As it will be seen in the example of Part III, the second intermediate text is close to the target code. STEP3 of the translators has to rewrite this text into some code, performing tasks such as registers allocation or unecessary statements removing. This step works in the same way as STEP2, evaluating the second term as a LISP form whose result is the generated code.

This system is currently developped. Several examples of compiler specifications [19,5,8] have been tested with this method.

To avoid inefficencies, STEP1 in addition of evaluating the semantic value of a source program, checks its correctness. Thus, no intermediate text is generated if the program is not valid. This verification has to be specified in the same time as semantic equations, using attributes.

From a practical point of view, the resulting translators seem, so far, to be efficient in time but rather expensive in space. The quality of the generated code depends on...the quality of the given specification. Let us consider now an example of such a specification.

III - EXAMPLE OF A COMPILER SPECIFICATION

The compiler presented here translates a source language with Algol-like blocks and arrays into a target language with addresses, registers, hexadecimal numbers, etc. Only some points of the specification are given since the goal is to show the specification method, not the compiler.

The implantation is a very classical one. Simple variables and arrays information vectors are allocated in a static memory area. Arrays are allocated at run-time, in a dynamic memory area, at the end of the memory.

The complete specification is given in appendix.

III.1 - Source Language Definition

The syntax of the source language is given below :

P → BLOCK

BLOCK → begin DL ; SL end

DL → DL ; D|D

D → integer ID|array ID [E:E]

SL → SL ; S|S

S → ID:=E|ID[E]:=E|BLOCK|begin SL end|
if C then S else S|while C do S

E → E+T|E-T|E*T|E/T|T

T → (E)|ID|ID[E]|NB

C → E=E|E≠E

NB and ID are considered to be terminal (their syntax will be dealt with by the scanner constructor).

III.1.1 - Semantics of expressions

As usually, the presentation of a data type associated with this language is made up of a list of names of type, a list of names of operation with their domains and co-domain and some axioms (or laws) which express the relationship between operations. In order to specify error cases, some restrictions are given, which describe the forbidden terms of the data type [1 2].
For the considered source language there are boolean (Bool), integer (Int), identifier of integer and array (Int-id, Array-id) among the names of type.

```
type Bool ;
op
  ( ) → Bool : true, false ;
  (Bool) → Bool : ^ ;
  (Bool,Bool) → Bool : and, or, impl, eq ;
axioms
  ...
end Bool ;
```

Bool and Int data types are well-known and their axioms are not given. Moreover, boolean operators are going to be used in a infixed way, with the usual rules for priority.

```
type Int ;
op
  (Int,Int) → Int : add, sub, mult, div ;
  (Int) → Bool : neg ;
  (Int,Int) → Bool : eq, neq ;
axioms
  ...
end Int ;
```

Let us call $\mathcal{V}$ the semantic function which returns the term of Int type corresponding to an arithmetic expression, and $\mathcal{B}$ the semantic function for conditions which returns a term of Bool type. Among the semantic equations, there are:

$\mathcal{V}[\![E+T]\!] = plus\ (\mathcal{V}[\![E]\!]\ ,\ \mathcal{V}[\![T]\!])$

$\mathcal{V}[\![(E)]\!] = \mathcal{V}[\![E]\!]$

$\mathcal{V}[\![NB]\!] = Int'NB'$

$\mathcal{B}[\![E1 \neq E2]\!] = neq\ (\mathcal{V}[\![E1]\!]\ ,\ \mathcal{V}[\![E2]\!])$

Note that constants of a type are enclosed between quotes, preceded by the name of their type.

It is important to point out that the operation names are not interpreted : we have no more information about 'plus' or 'neq' than what is specified in the presentation of 'Int'.

Semantics of identifiers is less straightforward to state since the properties of block structure must be described. A way to do that is to introduce some types environment (Env) and variables (Var) and to specify by means of axioms whic variable is designated by an identifier in a given environment. It must be noticed that an 'environment' here, is no more than a stack of identifiers which is modified by declarations and block's entries or exits.

```
type Array-id ;
op
     (Array-id, Array-id) → Bool:eq ;
axioms
     eq(Array-id 'C1', Array-id 'C2') = eq(C1, C2) ;
end  Array-id ;
type Int-id ;
...
```

```
end Int-id ;
type Id = union (Array-id, Int-id) ;
...
end Id ;
type Env ;
op
     ( ) → Env : current-env ;
     ( ) → Env : empty ;
     (Env) → Env : newblock, eraseblock ;
     (Env, Array-id) → Env : add-array ;
     (Env, Int-id) → Env : add-int ;
     (Env, Id) → Bool : is-in.a, is-in.i ... ;
axioms
     is-in.a(empty, id) = false ;
     is-in.a(newblock(e), id) = is-in.a(e,id) ;
     is-in.a(add-array(e,id1),id2) = if eq(id1, id2) then true
                                     else is-in.a(e, id2) ;
     is-in.a(add-int(e, id1), id2) = if eq(id1, id2) then false
                                     else is-in.a(e, id2) ;
     ...
     eraseblock(newblock(e)) = e ;
     eraseblock(add-array(e,id)) = erase(e) ;
     ...
end Env ;
```

The complete presentation of Env data type can be found in the appendix

```
type Var ;
op
     (Var, Var) → Bool : eq ;
     (Int-id, Env) → Var : des ;
     (Array-id, Env, Int) → Var : elt ;
     (Array-id, Env) → Int : lwb, upb ;
     (Var) → Int : val ;
axioms
     ...
```

```
eq(des(id, newblock(e)),des(id,e)) = true ;
eq(des(id1, add-int(e, id2)), des(id1, e)) = if eq(id1, id2)
                                              then false else true ;
...
end Var ;
```

The scope rules are expressed by preconditions on the term of type Var-For instance :

```
Pre(des,id,e) = is-in.i(e, id) ;
```

specifies that the term 'des(id, e)' can be written only if 'id' belongs to the environment 'e'.

Thus, the identifiers semantics is given by the following equations :

$$V[\![ID]\!] = val(des(Int\text{-}id\ 'ID',\ current\text{-}env)) ;$$

$$V[\![ID[E]]\!] = val(elt(Array\text{-}id\ 'ID',\ current\text{-}env,\ V[\![E]\!])) ;$$

(the preconditions, as it was said in Part II, had been checked using attributes)

III.1.2 - Notions of state and modification

So as to describe the semantics of statements (or declarations), we introduce a new name of type :

```
type Modif ;
```

The semantic value of a statement is a term of this data type. Let S be the corresponding semantic function. Among the semantic equations there are :

$$S[\![integer\ ID]\!] = int\text{-}decl(Int\text{-}id\ 'ID') ;$$

$$S[\![ID := E]\!] = int\text{-}assign(Int\text{-}id\ 'ID',\ V[\![E]\!]) ;$$

$$S[\![SL\ ;\ S]\!] = concat(S[SL],\ S[\![S]\!]) ;$$

$$S[\![begin\ DL\ ;\ SL\ end]\!] = concat^{3}(enter\text{-}block, S[\![DL]\!], S[\![SL]\!],\ exit\text{-}block) ;$$

where 'int-decl', int-assign', 'concat', 'enter-block' and 'exit-block' are some operations of the Modif data type :

```
op
  (Int-id) → Modif : int-decl ;
  (Int-id, Int) → Modif : int-assign ;
  (Modif, Modif) → Modif : concat ;
  ( ) → Modif : enter-block, exit-block ;
```

The intuitive meaning of this data type, is that a modification is a change of state. Thus, we have to define what is a state. Before giving this definition, it must be pointed out that Modif is the 'type of interest' of the algebraic data type associated with the programming language.

Definition :

Let <T, F, A> be the presentation of the algebraic data type associated with a language.

A state is a set S of formulas t=t', where t and t' are some terms of the data type, which satisfies the following properties :

i) $t = t \in S$ for all the terms t of the data type ;

ii) $t = t' \in S \Rightarrow t' = t \in S$;

iii) $t = t' \in S,\ t' = t'' \in S \Rightarrow t = t'' \in S$;

iv) for all f of F such as the co-domain of f is not Modif[2], if t and t' belongs to the domain of f :

$$t = t' \in S \Rightarrow f(t) = f(t') \in S.$$

A state satisfies the set of axioms A if it contains all the formulas obtained in substituting, in each axiom a of A, to all the free variables occuring in a, all the terms of <T, F, A> of the relevant domains.

Given a current state, a modification (an assignment for instance) removes some formulas from the state and add some new ones. In order to make easier the definitions of the modifications and the proofs of their representations, we suggest to use some primitives operations on modifications and states, which are common to all the language definitions.

The first operation is the application of a modification to a state :

$$\underline{\text{appl}}\ (m,S) = S'.$$

The second one is the adjonction of an axiom a to a state S : S+a is the smallest state which contains S and all the formulas obtained in substituting

---

(2) It is necessary to have this restriction in order to deal with procedure semantics without introducing some paradoxes [20].

to all the free variables in a, all the terms of the data type which are relevant.
The last one is a 'generalized substitution' which makes it possible to define all kind of assignement :

subst (f(λ),μ)

means, intuitively, that the value of function f, for all the terms equal to λ becomes μ.
subst is characterized by a rather complicated definition given below but it insures that no inconsistencies are introduced in the resulting state, even if there are some occurences of f in λ or μ. Thus it is possible to describe assignement for array's elements as well as for simple variables.
appl(subst(f(λ),μ),S) = {F|F[f'/f]∈S + f'(x) = if eq(x,λ) then μ else f(x)}

Using appl and subst, we can now give the definitions of the Modif data type operations of the considered Source Language :

definitions
    int-decl(id) = subst(current-env,add-int(current-env,id)) ;
    int-assign(id,i) = subst(val(des(id,current-env)),i) ;
    enter-block = subst(current-env,newblock(current-env)) ;
    exit-block = subst(current-env,eraseblock(current-env)) ;
    appl(concat(m1,m2),S) = appl(m2,appl(m1,S)) ;
    ...
    end Modif ;

III.2 - Example of a Target data type

The complete specification of the Target data type is given in appendix.
As a consequence of the kind of target language considered, there are among the name of types : hexadecimal numbers (Hexa), addresses , registers and condition codes (Cond-code). Among the name of modifications : load, store, compare, branch, branch under condition (cond-branch), branch under reverse condition (neg-branch) etc. We consider that the Hexa data type is predefined, with the same axioms as the Int data type :

```
type Hexa ;
op
     (Hexa,Hexa) → Hexa : plus, minus, mult, div ;
     (Hexa) → Bool : neg ;
     (Hexa,Hexa) → Bool : eq ;
axioms
     ...
end Hexa
```

It is important to notice that the Bool data type is here, in some way, an auxiliary type. It is needed to write conditionnal axioms, but the Target data type the result of a comparaison is a condition code, not a boolean.
The Address data type contains an indexing operation. From an address or a register one can get a 'Content', which is either an hexadecimal number or an address. The corresponding operations are named 'ca' and 'cr'.
The Cond-code data type is a bit more complicated : there are three constants in this type : 'lt', 'gt', 'eq'. These values are contained in a specific register, Register 'cond', which is modified only by the modification 'compare'. Thus, the definition of compare is :

compare(c1,c2) = subst(cc(Register'cond'), test(c1,c2))

where 'cc' returns the content of the register and test(c1,c2) is the result of the comparison.

III.3 - Specification of the implantation

In this part of the compiler specification, a representation is given for every operation name wich occurs in the R.H.S of the Source semantic equations. Only these representations are needed to perform the translation. If one want to get a correctness proof of the implantation, it becomes necessary to give a representation of all the operations occuring in the axioms (of all the operations, if the Source data type does not contain unecessary ones).
In this part of the paper, we are going to present only a part of the specification. Let us consider the Int data type first. It is going to be represented by the Hexa data type. This is specified in the following way :

```
type Int
     Hexa ;
     repr Int 'I' = Hexa 'CONVERT(I)' ;
     repr add(i,j) = plus(repr i, repr j) ;
     ...
     end Int ;
```

CONVERT is a meta-procedure which returns the text of the hexadecimal constant corresponding to I.

The data types Int-id, Array-id, Id and Env are not represented. The Var data type is represented by addresses, and the operation 'des' is represented by a constant address since the allocation of simple variables is static.

```
type Var ;
     Address ;
     repr des (Int-id 'ID', current-env) = Address 'SEARCH (ID)' ;
```

SEARCH is a meta-procedure. It looks for the address of identifier ID in a table TABLE, starting from the last added identifier. Its definition is given below :

```
     SEARCH (ID) = SEQ-SEARCH (ID, SIZE-1)
```

where

```
     SEQ-SEARCH (ID,i) = IF TABLE [i,1] = ID
                         THEN TABLE [i,2]
                         ELSE SEQ-SEARCH (ID,i-1)
```

Before going on in the specification, we specify the way the couples <identifier, address> are entered in TABLE. This is done by two functions : ALLOC1 for simple variables and ALLOC3 for arrays (3 words are allocated).

```
     ALLOC1 (ID) = (TABLE [SIZE] : = <ID, TOP (ALLOC-STACK)> ;
                    INCR (SIZE) ; INCR (TOP (ALLOC-STACK)))

     ALLOC3 (ID) = (TABLE [SIZE] : = <ID, TOP (ALLOC-STACK)> ;
                    INCR (SIZE) ; INCR3 (TOP (ALLOC-STACK)))
```

The three words allocated to an array are used in the following way : in the first one is the address of the beginning of the array ; the last ones contain the lower bound and the upper bound. Thus the representation of 'elt' is (index underflow or overflow are not tested) :

```
repr elt (Array-id 'ID', current-env, i) = indexing (ca (Address 'SEARCH (ID)'),
                minus (repr i, repr lwb (Array 'ID', current-env))) ;
where :
repr lwb (Array-id 'ID', current-env) =
                ca (indexing (Address 'SEARCH (ID)', Hexa '1') ;
```

Going on with the representation of Var data type :

```
repr val(v) = ca (repr v) ;
```

Among the modifications, we give the representation of the declarations, and of 'enter-block' and 'exit-block'.

```
repr int-decl (Int-id 'ID') = ## ALLOC1 (ID) ## nop ;
```

(the meta-statements, i.e compile-time computations, are noted between ##. 'nop' is the 'no operation' target statement).

```
repr array-decl (Array 'ID', i, j) = ## ALLOC3 (ID) ;
          A : = SEARCH (ID) ##
          seqcomp^3 (store (repr i, indexing (Address 'A', Hexa '1')),
                    store (repr j, indexing (Address 'A', Hexa '2')),
                    load (minus (cr (Register 'free'),
                                 plus (minus (repr j, repr i), Hexa '1')),
                          Register'free'),
                    store (cr (Register 'free'), Address 'A')) ;
```

(it should be verified that $j-i \geq 0$).

Register 'free' points on the first free address before the last allocated array Dynamic allocation is done in a decreasing way in the memory addresses.

```
repr enter-block = ## A : = TOP (ALLOC-STACK) ;
          PUSH (ALLOC-STACK, A+1), PUSH (TAB-STACK, SIZE) ##
          store (cr (Register 'free'), Address 'A') ;
```

It is a bit confusing but not really complicated. Before entering a block, at compile-time, the address following the previous block allocations is push on a stack and the allocation of the new block will start from it (see ALLOC1) and the SIZE of TABLE is saved. At run-time, the pointer in the dynamic area is saved.

```
repr exit-block = ## POP (ALLOC-STACK) ; A : = TOP (ALLOC-STACK) ;
          SIZE : = TOP (TAB-STACK) ;
          POP (TAB-STACK) ## load (ca (Address 'A'), Register 'free') ;
```

It may be seen that the specification of the implementation becomes very systematic and formal. But the main advantage is the ability to check up such implementations. In several previous examples, some errors in the specification come to light during the proof process.

IV - CORRECTNESS PROOF OF IMPLEMENTATIONS

The correctness proof methodology is based on the distinction of 'modifiable' operations from classical operations. Modifiable operations are operations which appear in a subst, in some modification definition.

There must not be axioms on these operations since subst could introduce some inconsistencies with them. Besides, to be correct, the representation of such operations must verify some strong properties : if two such operations are represented by the same operation of the Target data type, then, there must be no overlapping between the domains of these representations (if it were the case, it would be possible to modify two Source operations by the same Target modification).
The proof method is outlined below : first it must be verified that the implementation satisfies the above property for modifiable operations ; then, the Source data type, without the modifiable operations and the modifications, is considered,and it must be proved that its representation into the Target data type is correct in the sense of [10] (the axioms are kept) ; the last step is the correctness proof of the representation of modifications.
The representation of a modification is correct if :

- it keeps unchanged the representation invariants which arised in the previous proof (Source data type without modifications)
- if the modification is defined by a subst $(f(\lambda),\mu)$, then the representation embodies a modification which can be proved to be equivalent to subst (repr $f(\lambda)$, repr μ)
- the other modifications occuring in the representation are "without side-effect", i.e without effect on the Source data type. They do not modify the representation of any Source operation.

Such a proof is very long, but it is built systematically, and the various steps are not very difficult. A complete proof is given in [20].

ACKNOWLEDGMENTS

Theoretical as well practical aspects of this work have been studied in collaboration with Ph. DESCHAMP, M. MAZAUD and C. PAIR. P. BOULLIER and B. LORHO's previous works made it possible the effective developpment of the system. P. DERANSART's knowledge of LISP was very useful.

REFERENCES

[1] P. BOULLIER. Le système SYNTAX, Manuel d'utilisation, Groupe Langages et Traducteur, IRIA, 1977.

[2] P. BOULLIER. Automatic Syntatic Error Recovery for LR-Parsers, 5th Annual III Conference - Guidel, Mai 1977.

[3] P.D. MOSSES. Mathematical Semantics and Compiler Generation. Ph.D. Thesis, Université d'Oxford, 1975.

[4] P.D. MOSSES. SIS, a Compiler-Generator System using Denotational Semantics Dept. of Computer Science, Université d'Aarhus, June 1978.

[5] M.C. GAUDEL. A formal Approach to Translator Specification, IFIP Congress, Toronto, 1977.

[6] M.C. GAUDEL, Ph. DESCAMP, M. MAZAUD. Semantics of Procedures as an Algebraic Abstract Data Type, Rapport Laboria n° 334 - 1978.

[7] M.C. GAUDEL, C. PAIR. Construction de Compilateurs basée sur une Sémantique Formelle. Acte des journées francophones sur la certification du logiciel, Genève, 1979.

[8] M.C GAUDEL, C. PAIR. The Use of a Formal Semantics to Produce and Prove Compilers. International Workshop on the Semantics of Programming Language. Bad Honnef. March 1979.

[9] F. Lockwood MORRIS. Advice on Structuring Compilers and Proving them Correct. P.O.P.L. 1973 .

[10] J.V. GUTTAG, E. HOROWITZ, D.R. MUSSER. Abstract Data Types and Software Validation. CACM. December 1978.

[11] J.W. THATCHER, E. WAGNER, J.B. WRIGHT. An Initial Algebra approach to the Specification, Correctness, and Implementation of Abstract Data Type. In Current Trends in Programming Methodology IV (R. Yeh. Ed). Prentice Hall 1979.

[12] M.C. GAUDEL. An introduction to Algebraic Abstract Data Type. Lecture notes of the 2nd advanced course on Computing System Reliability. IRIA, Sept 1979.

[13] M. GORDON, R. MILNER, C. WADSWORTH - Edinburgh LCF. Internal Report CSR-11-77 University of Edinburgh, Sept 1977.

[14] D.R. MUSSER. A Data Type Verification System based on Rewrite Rules. 6th Texas Conference on Computing System, Austin, Nov. 1977.

[15] D.R. MUSSER. Abstract Data Type Specification in the AFFIRM System. IEEE Transactions on Software Engineering (to appear).

[16] D.E. KNUTH. Semantics of Context-free Languages. Math. Systems Theory, Vol. n°5, n°1, 1971.

[17] B. LORHO. De la Définition à la Traduction des Langages de Programmation: la méthode des attributs sémantiques. Thèse d'Etat - Toulouse, 1974.

[18] B. LORHO. DELTA : manuel d'utilisation, Groupe Langages et Traducteurs, IRIA, Nov. 1977.

[19] M.C. DENDIEN-GAUDEL. Applications des Structures de Données à la description et à la preuve de Traducteurs. Congrès AFCET Informatique, Gif-sur-Yvette, Nov. 1976.

[20] M.C. GAUDEL. Thesis, University of Nancy (France), March 1980.

APPENDIX 1

DEFINITION OF THE SOURCE LANGUAGE.

```
type Bool ;
op
  () -> Bool : true, false ;
  (Bool) -> Bool : ^ ;   ****it is the 'not' operator****
  (Bool, Bool) -> Bool : and, or, impl, eq;
axioms
  .
  .
end Bool ;

type Int ;
op
  (Int, Int) -> Int : add, sub, mult, div ;
  (Int) -> Bool : neg ;
  (Int, Int) -> Bool : eq, neq ;
axioms
  .
end Int ;

type Array-id ;
op
  (Array-id, Array-id) -> Bool : eq ;
axioms
  eq(Array-id'C1', Array-id'C2') = eq(C1, C2) ;
end Array-id ;

type Int-id ;
op
  (Int-id, Int-id) -> Bool :eq ;
axioms
  eq(Int-id'C1', Int-id'C2') = eq(C1, C2) ;
end Int-id ;

type Id = union(Array-id, Int-id) ;
op
  (Id, Id) -> Bool :eq ;
axioms
  eq(Int-id'C1', Array-id'C2') = eq(C1, C2) ;
  eq(Array-id'C1', Int-id'C2') = eq(C1, C2) ;
end Id ;
```

```
type Env ;
op
  () -> Env : empty ;
  () -> Env : current-env ;
  (Env) -> Env : newblock, eraseblock ;
  (Env, Array-id) -> Env : add-array ;
  (Env, Int-id) -> Env : add-int ;
  (Env, Id) -> Bool : is-in.a, is-in.i, is-local ;
axioms
  is-in.a(empty, id) = false ;
  is-in.a(newblock(e), id) = is-in.a(e, id) ;
  is-in.a(add-array(e, id1), id2) = if eq(id1, id2) then true
                                    else is-in.a(e,id2) ;
  is-in.a(add-int(e, id1), id2) = if eq(id1, id2) then false
                                  else is-in.a(e, id2) ;

  is-in.i(empty, id) = false ;
  is-in.i(newblock(e), id) = is-in.i(e, id) ;
  is-in.i(add-array(e, id1), id2) = if eq(id1, id2) then false
                                    else is-in.i(e, id2) ;
  is-in.i(add-int(e, id1), id2) = if eq(id1, id2) then true
                                  else is-in.i(e, id2) ;

  is-local(empty, id) = false ;
  is-local(newblock(e), id) = false ;
  is-local(add-array(e, id1), id2) = if eq(id1,id2) then true
                                     else is-local(e, id2) ;
  is-local(add-int(e, id1), id2) = if eq(id1, id2) then true
                                   else is-local(e, id2) ;

  eraseblock(empty) =empty ;
  eraseblock(newblock(e)) = e ;
  eraseblock(add-array(e, id)) = eraseblock(e) ;
  eraseblock(add-int(e, id)) = eraseblock(e) ;
restrictions
  Pre(add-array, e, id) = ^ is-local(e, id) ;
  Pre(add-int, e, id) = ^ is-local(e, id) ;
end Env ;

type Var ;
op
  (Var, Var) -> Bool :eq ;
  (Int-id, Env) -> Var : des ;
  (Array-id, Env, Int) -> Var : elt ;
  (Array-id, Env) -> Int : lwb, upb ;
  (Var) -> Int : val ;
axioms
**** eq is an equivalence relation and besides : ****
  eq(des(id1, e), des(id2, e)) = eq(id1,id2) ;
  eq(des(id, newblock(e)), des(id, e)) = true ;
  eq(des(id1, add-int(e, id2)), des(id1, e)) = ^eq(id1, id2) ;
  eq(elt(id1, e, i), elt(id2, e, j)) = eq(id1, id2) and eq(i, j) ;
  eq(elt(id,newblock(e), i), elt(id, e, i)) = true ;
  eq(elt(id1, add-array(e, id2), i), elt(id, e, i)) = ^eq(id1, id2) ;
  eq(des(id1, e1), elt(id2, e2, i)) = false ;
restrictions
  Pre(des, id, e) = is-in.i (e, id) ;
  Pre(elt, id, e, i) = is-in.a(e, id) ;
  Pre(upb, id, e) = is-in.a(e, id) ;
  Pre(lwb, id, e) = is-in.a(e, id) ;
  **** arrays underflow and overflow specifications ****
  neg(sub(i, lwb(id, e))) => Failure(elt, id, e, i) ;
  neg(sub(upb(id, e), i)) => Failure(elt, id, e, i) ;
end Var ;
```

```
type Modif ;
op
  (Int-id) -> Modif : int-decl ;
  (Array-id, Int, Int) -> Modif : array-decl ;
  (Int-id, Int) -> Modif : int-assign ;
  (Array-id, Int, Int) -> Modif : array-assign ;
  () -> Modif : init, enter-block, exit-block ;
  (Bool, Modif, Modif) -> Modif : cond ;
  (Bool, Modif) -> Modif : loop ;
  (Modif, Modif) -> Modif : concat ;
definitions
  int-decl(id) = subst(current-env, add-int(current-env, id)) ;
  array-decl(id, i, j) = concat(subst(current-env, add-array(current-env, id)),
                                subst(<lwb(id,current-env),upb(id,current-env)>
                                      <i, j>) ) ;
  **** subst(<f(x), g(y)>, <v, w>) is a concurrent substitution ****
  int-assign(id, i) = subst(val(des(id, current-env)), i) ;
  array-assign (id, i, j) = subst(val(elt(id, current-env, i)), j) ;
  init = subst(current-env, empty) ;
  enter-block = subst(current-env, newblock(current-env)) ;
  exit-block = subst(current-env, eraseblock(current-env)) ;
  **** @ means 'belongs to ' ********************
  b=true @ S => appl(cond(b, m1, m2), S) = appl(m1, S) ;
  b=false @ S => appl(cond(b, m1, m2), S) = appl(m2, S) ;
  b=true @ S => appl(loop(b, m), S) = appl(loop(b, m), appl(m, S)) ;
  b=false @ S => appl(loop(b, m), S) = S ;
  appl(concat(m1, m2), S) = appl(m2, appl(m1, S) ;
restrictions
  Pre(int-assign, id, i) = is-in.i(current-env, id) ;
  Pre(array-assign, id, i, j) = is-in.a(current-env, id) ;
  neq(sub(i, lwb(id, current-env))) => Failure(array-assign, id, i, j) ;
  neq(sub(upb(id, current-env), j)) => Failure(array-assign, id, i, j) ;
end Modif ;

****************SEMANTIC EQUATIONS*************************************

        M : P -> term of data type Modif
    M(begin DL ; SL end) = concat.3(init, S(DL), S(SL)) ;
        S : D,S,DL,SL -> term of data type Modif
    S(DL;D) = concat(S(DL), S(D)) ;
    S(integer ID) = int-decl(Int-id'ID') ;
    S(array ID[E1:E2]) = array-decl(Array-id'ID',V1(E1),V1(E2)) ;
    S(SL;S) = concat(S(SL), S(S)) ;****sorry for the S(S) !****
    S(ID:=E) = int-assign(Int-id'ID', V(E)) ;

    S(ID[E1]:=E2) = array-assign(Array-id'ID', V(E1), V(E2)) ;
    S(begin DL;SL end) = concat.4(enter-block,S(DL),S(SL),exit-block) ;
    S(begin SL end) = S(SL) ;
    S(if C then S1 else S2) = cond(B(C), S(S1), S(S2)) ;
    S(while C do S) = loop(B(C), S(S)) ;

        V,V1 : E,T -> term of Int data type
    V(E+T) = add(V(E), V(T)) ;
   *****it is the same thing for the other operations*****
    V((E)) = V(E) ;
(1) V(ID) = val(des(Int-id'ID',current-env)) ;
(2) V(ID[E]) = val(elt(Array-id'ID', current-env, V(E))) ;
    V(NB) = Int'NB' ;
    *****V1 is defined by the same equations as V, but equations*******
    *****(1) and (2) where current-env is replaced by           ********
    *****erase(current-env).Indeed in an array declaration the *********
    *****bounds must be evaluated in the embedding environment *********
        B : C -> term of Bool data type
    B(E1=E2) = eq(V(E1), V(E2)) ;
    B(E1#E2) = neq(V(E1), V(E2)) ;
```

APPENDIX 2

```
**********TARGET DATA TYPE **********

type Hexa ;
op
  (Hexa, Hexa) -> Hexa :plus,minus,mult,div ;
  (Hexa, Hexa) -> Bool : eq ;
axioms
  ...
end Hexa ;

type Address ;
op
  (Address, Hexa) -> Address : indexing ;
  (Address, Address) -> Bool : eq ;
axioms
  indexing(a, Hexa'0') = a ;
  indexing(indexing(a, h1), h2) = indexing(a, plus(h1, h2)) ;
  eq(Address'C1', Address'C2') = eq(C1, C2) ;
end Address ;

type Register ;
op
  (Register,Register) -> Bool : eq ;
axioms
  eq(Register'C1', Register'C2') = eq(C1, C2) ;
end Register ;

type Content = union (Hexa, Address) ;
op
 (Address) -> Content : ca ;
 (Register) -> Content : cr ;
restrictions
 Pre (cr, r) = ^ eq(r, Register'cond') ;

type Cond-code ;
op
  () -> Cond-code : lt,gt,eq ;
  (Content, Content) -> Cond-code : test ;
  (Register) -> Cond-code : cc ;
  (Cond-code) -> Cond-code : sym ;
axioms
  sym(eq) = eq ;
  sym(gt) = lt ;
  sym(lt) = gt ;
  test(h1, plus(h2, Hexa'1')) = if eq(h1, h2) then lt
                                else if eq(h1,plus(h2,Hexa'1'))
                                     then eq
                                     else test(h1,h2) ;
  test(h1, minus(h2, Hexa'1')) = if eq(h1, h2) then gt
                                 else if eq(h1,minus(h2,Hexa'1'))
                                      then eq
                                      else test(h1, h2) ;
  test(a1, indexing(a2, Hexa'1')) = if eq(a1, a2) then lt
                                    else if eq(a1,indexing(a2,Hexa'1'))
                                         then eq
                                         else test(a1, a2) ;
  test(c1, c2) = sym(test(c2, c1)) ;
restrictions
  Pre (cc, r) = eq(r, Register'cond') ;
end Cond-code ;
```

```
type Label ;
op
   (Label, Label) -> Bool :eq ;
axioms
   eq(Label'C1', Label'C2') = eq(C1, C2) ;
end Label ;

type Modif ;
op
   (Content, Register) -> Modif : load ;
   (Content, Address) -> Modif : store ;
   (Content, Content) -> Modif : compare ;
   (Label, Modif) -> Modif : labelled ;
   (Modif, Label) -> Bool : exit,entry ;
   (Label) -> Modif : branch ;
   (Label, Cond-code) -> Modif : cond-branch, neg-branch ;
   (Modif, Modif) -> Modif : seqcomp ;
   () -> Modif : nop ;
   (Modif, Content) -> Content : side-effect ;
   (Modif, Cond-code) -> Cond-code : c.side-eff, nc.side-eff ;

axioms
   entry(load(c, r), l) = false ;
   entry(store(c, a), l) = false ;
   entry(compare(c1, c2), l) = false ;
   entry(branch(l1), l2) = false ;
   entry(cond-branch(l1, ccond), l2) = false ;
   entry(neg-branch(l1, ccond), l2) =false ;
   entry(labelled(l1, m), l2) = if eq(l1, l2) then true
                                              else entry(m, l2) ;
   entry(seqcomp(m1, m2), l) = entry(m1, l) or entry(m2 ,l) ;
   entry(nop, l) = false ;

   exit(load(c, r), l) = false ;
   exit(store(c, a), l) = false ;
   exit(compare(c1, c2), l) = false ;
   exit(branch(l1), l2) = eq(l1, l2) ;
   exit(cond-branch(l1, ccond), l2) = eq(l1, l2) ;
   exit(neg-branch(l1,  ccond), l2) = eq(l1, l2) ;
   exit(labelled(l1, m), l2) = if eq(l1,l2) then false
                                            else exit(m, l2) ;
   exit(seqcomp(m1, m2), l) = if entry(m1, l) or entry(m2, l)
                              then false
                              else exit(m1, l) or exit(m2, l) ;
   exit(nop, l) = false ;

definitions
   load(c, r) = subst(cr(r), c) ;
   store(c, a) = subst(ca(a), c) ;
   compare(c1,c2) = subst(cc(Register'cond'), test(c1, c2)) ;

   ^exit(m, l) => appl(labelled(l, m), S) = appl(m, S) ;

   appl(seqcomp(labelled(l, m1), m2), S) =
                appl(labelled(l,seqcomp(m1, m2)), S) ;

   ^exit(m1, l) => appl(seqcomp(m1, m2), S) =appl(m2, appl(m1,S)) ;

   appl(seqcomp.3(branch(l), m1, labelled(l, m2)), S) =
                   appl(labelled(l, m2), S) ;
```

```
   ccond=cc(Register'cond') @ S =>
                    appl(seqcomp.3(cond-branch(l,ccond),m1,labelled(l,m2)), S)
                    = appl(labelled(l, m2), S) ;

   ^ccond=cc(Register'cond') @ S =>
                    appl(seqcomp.3(cond-branch(l,ccond),m1,labelled(l,m2)), S)
                    = appl(seqcomp(m1, labelled(l, m2)), S) ;
***rules for neg-branch are the reverse ones ****************
   let m=labelled(l1,seqcomp.5(m1,cond-branch(l2,ccond),m2,branch(l1),
                                labelled(l2,m3)))
   For all l in Label, exit(m1,l)=false and exit(m2,l)=false
   then the two following rules hold :
   cc(Register'cond')=ccond @ appl(m1, S) =>
       appl(m, S) = appl(seqcomp(m1, labelled(l2, m3)), S) ;
   ^cc(Register'cond')=ccond @ appl(m1, S) =>
       appl(m, S) = appl(m, appl(seqcomp(m1, m2), S)) ;


   appl(nop, S) = S ;
   ca(h.side-effect(m,a)) = h.side-effect(m, ca(a)) ;

   cond-branch(l,c.side-eff(m,ccond)) = seqcomp(m,cond-branch(l,ccond)) ;
   cond-branch(l,nc.side-eff(m,ccond)) = seqcomp(m,neg-branch(l,ccond)) ;

   neg-branch(l,c.side-eff(m,ccond)) = seqcomp(m,neg-branch(l,ccond)) ;
   neg-branch(l,nc.side-eff(m,ccond)) = seqcomp(m,cond-branch(l,ccond)) ;

************END OF THE TARGET DATA TYPE**************************
```

APPENDIX 3

SPECIFICATION OF THE IMPLEMENTATION

```
********Only the operations and types occuring in the semantic equations*****
*********  are represented . ***********************************************

type Bool ;
   Cond-code ;
end Bool ;

type Int ;
   Hexa ;
   repr Int'I' = Hexa'CONVERT(I)' ;
   repr add(i, j) = plus(repr i, repr j) ;
   repr sub(i, j) = minus(repr i, repr j) ;
   repr mult(i, j) = mult(repr i, repr j) ;
   repr div(i, j) = div(repr i, repr j) ;
   repr eq(i, j) = side-effect.c( compare(repr i, repr j), eq) ;
   repr neq(i, j) = side-effect.nc( compare(repr i, repr j), eq) ;
end Int ;

*******Int-id, Array-id, Id and Env are not represented . *****************

type Var ;
   Address ;
   repr des(Int-id'ID', current-env) = Address'SEARCH(ID)' ;
   repr des(Int-id'ID', eraseblock(current-env)) = Address'SEARCH1(ID)' ;
   repr elt(Array-id'ID', current-env, i) = indexing( ca(Address'SEARCH(ID)'),
                                              minus(repr i,
                                                    repr lwb(Array-id'ID',
                                                             current-env))) ;
   repr lwb(Array-id'ID', current-env) = ca(indexing(Address'SEARCH(ID)',
                                                     Hexa'1')) ;
   repr elt(Array-id'ID', eraseblock(current-env), i) = indexing(
                              ca(Address'SEARCH1(ID)'),
                              minus(repr i, repr lwb(Array-id'ID',
                                                     eraseblock(current-env)))) ;
   repr lwb(Array-id'ID', eraseblock(current-env)) = ca(indexing(
                              Address'SEARCH1(ID)', Hexa'1')) ;
   repr val(v) = ca(repr v) ;
end Val ;
```

```
type Modif ;
   repr int-decl(Int-id'ID') = # ALLOC1(ID) # nop ;
   repr array-decl(Array-id'ID', i, j) = # ALLOC3(ID); A:=SEARCH(ID) #
                        seqcomp.4(store(repr i, indexing(Address'A', Hexa'1')),
                                  store(repr j, indexing(Address'A', Hexa'2')),
                                  load(minus(cr(Register'free'),
                                             plus(minus(repr j,repr i),Hexa'1'),
                                       Register'free'),
                                  store(cr(Register'free'), Address'A') ) ;
****** It should be verified that j-i+1>0 . ********************************
   repr int-assign(Int-id'ID', i) = store(repr i, Address'SEARCH(ID)') ;
   repr array-assign(Array-id'ID', i, j) = # A:=SEARCH(ID) ; R:=GENREG #
                        compseq.6(load(repr i, Register'R'),
                                  compare(cr(Register'R'),ca(indexing(Address'A',
                                                             Hexa'2'))),
                                  cond-branch(Label'overflow', gt),
                                  compare(cr(Register'R'),ca(indexing(Address'A',
                                                             Hexa'1'))),
                                  cond-branch(Label'underflow', lt),
                                  store(repr j, indexing(ca(Address'A'),
                                                         minus(cr(Register'R'),
                                                               ca(indexing(
                                                                  Address'A',
                                                                  Hexa'1'))))))) ;
   repr enter-block = $ A := TOP(ALLOC-STACK) ; PUSH(ALLOC-STACK, A+1) ;
                        PUSH(TAB-STACK, SIZE) $
                      store( cr(Register'free'), Address'A') ;

   repr exit-block = $ POP(ALLOC-STACK) ; A := TOP(ALLOC-STACK) ;
                       SIZE := TOP(TAB-STACK) ; POP(TAB-STACK) $
                     load( ca(Address'A'), Register'free') ;

   repr init = $ ALLOC-STACK := EMPTY ; PUSH(ALLOC-STACK, 1) ; SIZE:= 1 ;
                 TAB-STACK := EMPTY $ load( Address'max', Register'free') ;
```