AUTOMATIC PROGRAMMING USING ABSTRACT DATA TYPES

Gerard Guiho

Universite Paris Sud LRI - Bat. 490 91405 ORSAY - FRANCE

ABSTRACT

In this paper we first try to characterize one meaning of automatic programming. We consider it to be one part of the Programming environment related to Artificial Intelligence techniques. We then illustrate an automatic programning process, on a simple example, using an Abstract Data Type theory to which we add the not ion of schemes which are particularly useful in program derivation from Abstract Type decomposition. We conclude that all the concepts treated in this paper must be contained in one way or another in any automatic programming system. However this necessitates further study in such theoretical fields as Abstract Data Type Theory, Specification languages, Theorem provers or proof cheekers and rule rewriting systems.

I GENERALITIES

I.I. In my opinion the concepts implied in the words "Automatic Programning" are not very precise and may even seem completly unrealistic. However many people, like myself may consider that even if they seem vague and sometimes unrealistic they appear worthwhile for studying.

In the first part of my paper, I will try to specify the definition of these concepts, created by the juxtaposition of the two words "Automatic" and "Programming".

In Webster's dictionary, we can see that "automatic" can be the obtention of something which can be produced without thinking, by habit or reflex. In this case, Automatic Programming could be considered as a kind of programming methodology, which is sufficient to follow to obtain a good program. This can only be applied in areas where the programing process is repetitive enough that a method can grasp the mechanical aspect. In this way. in the business-oriented programming field, some program generators or methodologies can be considered as automatic programming.

A second meaning to the word automatic concerns one action which is done by itself. We could then look for a programming technique which is entirely produced by any kind of mechanism. This is a more Artificial Intelligence approach. In this sense we could consider building a system which would receive the specification of a problem as input and which will give a program as output. This field is often called Program Synthesis. The input specification can be formal specifications (as logic) [ll[2] , Examples [3] [4] f 5] or Natural Language. Taking into account the research which has been done on this subject during the last five years and my own personal experience, I think that this approach is only possible for restricted fields of application and toy problems. In other cases the specification language is close to a programming language and it is more a compiling technique than automatic programming. I am therefore convinced that this approach can be useful locally but that programming, in the general sense, will never be completly automatic.

A third sense to the word automatic will also help us. It characterizes one action which is done using automatic equipment. This dictionary definition is in fact recursive ! ! That leaves us now to consider that it is not the action of programming which has to be automatic but the equipment which could help do this action. Lastly we can interpret "Automatic Programming" as the programming action using a computer, which allows us to write programs in the best way possible". That does not imply that the whole programming process has to be automatic.

Unfortunately,this definition is too general. If any system used during the programming process is relevant to automatic programming then most of the programmers are in fact using automatic programming techniques without knowing it. Is a text editor, a compiler, a linker, ... relevant to automatic programming ? And is there any difference between "Automatic Programming" and "Programming Environment" ?

In fact I consider that automatic programming is a part of what is called programming environment. It consists of the tools which are the most advanced in the programming environment and which are directly related to the programmer. We are very close here to an Artificial Intelligence paradigm because the action of programming can be considered as one of the most difficult and intelligent action a human can do. Sofisticated tools which could help the programmer during this powerful action may be relevant to the Artificial Intelligence field.

However, even if the line between Artificial Intelligence and classical Computer Science is not really precise in Automatic Programming, I consider that it is a domain in which many different

Artificial Intelligence techniques can be extremely useful. This must be a challenge for Artificial Intelligence people. It is not a coincidence if the first and most powerful programming environments came from the Artificial Intelligence community, (Interlisp, Maclisp...)

1.2. I will summarize some fields of Artificial Intelligence and just give some areas where they could be profitable in automatic programming.

- Natural Language Understanding—This subfield could be very useful, not in the command to be given to a system but to help the programmer in the various commands and utilities of the programming environment. For an experienced programmer, the the commands he knows have to be very short. They are generally keys. The problem is that when we want to explore some new possibilities, we do not know precisely the keys or the successions of keys to use for these new possibilities. Looking through the documentation (even in line) is sometimes boring. The "a propos" command under emacs is useful but is only a key word search. The development of a natural language interface would allow a sophisticated help system which would then be very effective.

- Expert Systems—I do not think an expert system can be constructed at the present in the program-creating process because it is too difficult and may even be outside of classical expert systems approach. However, in many parts of the programming process an expert system viewed as an assistant can be used. These systems have to be separate tools :

. The organisation of large programs. How to find some information in a large base of programs or data types ? What has been done until now and what would be the most suitable to do now etc...? It is a knowledge-based oriented system [6].

. The process of transforming programs [7] .

. Restricted areas. When the field is very small and the derivation of the program from the specification very easy, it would be fruitful to design some small expert systems.

. The validation of programs using test sets. The generation of the test set for programs is never completly satisfactory and is a difficult art. Testing a program can be relevant to Expert System techniques.

- Theorem Proving—Testing programs are not sufficient and in the future most of the programs will have to be proven, at least partially. Many interactive systems will have to be designed in order to help the programmer prove the correctness of this work. This domain is highly related to Theoretical Computer Science because most of the concepts in languages or programs have not yet been sufficiently studied to be used in practive for real programs. Some systems as GYPSY [8] , STP [9] , FORMEL [10] are milestones towards this approach.

The rest of this paper will try to show that even for small examples, the proof can be long and the material involved very sophisticated.

- Other techniques—Many other aspects which only use heuristics can be very useful in the programming process. I will just give some of them here but this is not exhaustive.

-Help during the programming process—Proposing program schemes, data decomposition, programs which are "close" to the one the programmer is actually doing etc...

- Intelligent display of all the information needed during the programming process (a screen with a multiple-window oriented editor).

- Organisation for work scheduling.

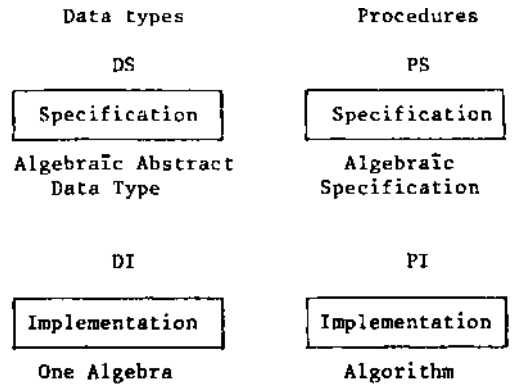
II PROGRAM CONSTRUCTION

We will.now describe one proposal for a program construction technique. It is not aimed to be entirely general but shows many concepts which, in one form or another, are necessary in every future automatic programming system.

It is now beginning to be admitted that a library which is useful in the programming process must contain both data and programs. All these objects have to be encapsulated in modules. Some of these modules involve descriptions of powerful data types with basic operators ; some others involve sets of programs with one common functionality.

Ail these modules contains two parts. One which shows that which is visible outside of the module, which attempts to describe the specification of the manner in which to use it. The second part describes how effectively the elements of the modules are implemented. This is one of the most important aspects of ADA with the PACKAGE and PACKAGE BODY parts. Generally the description part (which we will call the specification part) only contains the profile of the operations which are visible outside the module and few other helpful things for type cheeking and documentation. However, even if this aspect constitutes a major improvement, it does not allows us to produce proofs.

The nature of the objects which have to be grasped in our system can be represented as :



where

1. Data Type Specification (DS)

We use Algebraic Abstract Data Types in an extended manner which will be described in sectionIII. According to the theory, an Algebraic Abstract Data Type specification denotes a class of Algebra Alg_Σ and in our theory this class has one initial algebra T_Σ. (ie for each algebra A in Alg_Σ there exists one unique morphism h : T_Σ+A.

2. Data Implementation (DI)

Generally the Data Implementation is not vi'. sible to the user (and it must not be visible). We will consider that the Data Implementation represents one of the algebra in Alg_Σ denoted by the abstract type specification. The fact that there exists one morphism between T_Σ (which is a particular Algebra of Terms) and A helps us prove properties or theorems on the type using T_Σ.. These properties will be properties of any implementation. Of course the correctness of the implementation has to be provenin the same ways that many properties concerning abstract specification. That is done once for all and could be considered as the responsability of the data type designer. Note that our class of algebra is the class of finitely generated algebras so that we can use term rewriting and structural induction for producing proof.

3. Procedure specification (PS)

In a first approximation we will consider specifications in an algebraic manner. This will be easier for proofs but it may lead to specifications which are not really readable. I consider that there does not exist for the moment an effective, convenient specification language. If one such language would exist, its semantic would have to be expressed algebraically, but for our purposes here I have chosen to express it directly in its algebraic form.

4. Procedure implementation (PI)

First we need a programming language with a well-defined semantic in order to produce proofs. Section IV will describe such a toy language.

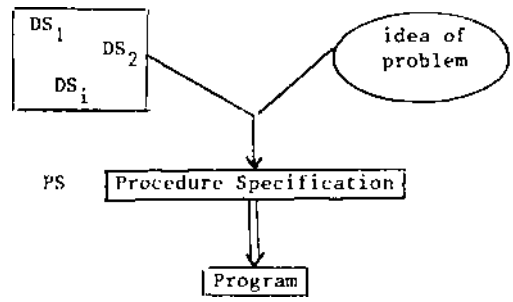
Given a program P written in such a language and proving its correctness may not be sufficient. These properties are proven in fact, in an extension of T , the initial term algebra : T_Σ + P and the program will actually be used in an algebra A + P. The fact that there exists an horaomorphism h : T_Σ-+A does not prove that it can be naturally extended to an homomorphism n : T_Σ + P --> A + P. This has to be proven again and it can be done either :

- By proving it on each program !!
- By restricting the form of programs or by directing the program construction such that any morphism can be extended.
- By using monomorphic abstract types.

5. Program construction

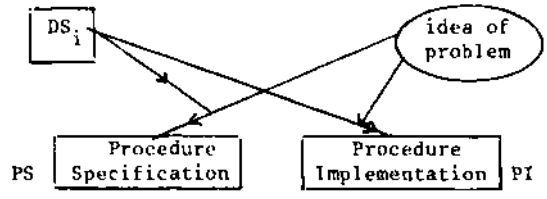
Two main methods can be used in building procedures concerning a problem we have in mind or which is expressed in a natural language manner :

a. Knowing a family of Data Types Specification, build a procedure specification using any intuitive method and then derive a program.



The problem whith this method is that even for short specifications, the risk of error is high (it may be even higher than directly deriving the program if the specification language is obscure). Subsequently the program will be wrong and if the procedure specification represents in some sense a "contract", this could be dangerous. One other problem is that only when the program is effectively tested that some errors will occur. It might not be easy to see where they correspond to a mistake in the specification. The risk is that the programmer will directly change the program !! One could say that we have some executable specifications but the other risk is that the program might easily follow the specification and could be highly uneffective.

b. Knowing a family of Data Type Specifications, build a procedure specification, then build a program separately and prove the correctness of the program versus the specifications.



Proof

Here we have more chance to make errors in the two constructions but the errors will not be necessarily the same. The proof mechanism will help us to correct the two parts. In my experience I always made aproximately the same number of mistakes in writing programs or specifications. Trying to prove the correctness always leads me to reconsider both. The risk is that if the specification is too far from the implementation (highly non executable specification for instance) the proof could be very difficult.

One other way would be to extract automatically the test set from the specification in order to test the program. [11],

This is the type of program construction tech-

nique we wil use in further sections with a systematic methodology for building programs. It could help us reduce the amount of errors when we try to express our intui tion.Wewill use one toy example all along (the gcd problem) because it is sufficiently complex, the concepts significant and sufficiently short to be shown in one paper.

III ALGEBRAIC ABSTRACT DATA TYPeS

We will use the now classical theory of abstract types which is directly inspired from ADJ [12] , Goguen [1 3l [14] , Broy & Wirsing [1 5] , complemented by work from Bidoit [16] , Kaplan [17] , myself, Boisson and Pavot [18] . Our addenda concern principally the presentation of type, the use of positive conditional axioms and the error mechanism.

The schemes are direct extensions of the decomposition schemes of C. Gresse [19] (same proceedings).

An abstract type is given by :

1. A signature represented by a set S of Sorts and a set ∑. of symbols with an arity in (S) . This differs from the usual theory where the arity belongs to S). We use, as Goguen does, overloading and coercion extensively. The notation of operators is similar to OBJ (\2\ .

2. A set of positive conditional axioms.

These axioms (as in Goguen [14])contains the sort in which the equation has to be considered. The operators are raultioperators and can have more than one output. In fact, their interpretation is a function from the domaines of their input to the union of domaines of their output (which have to be disjointed). The operators have to be total on the ground terms but can be partial on terms with variables. Using some kinds of presentation and with some properties, it can be shown that there can exist one initial algebra of terms in the class of specified algebras. More details are available in Boisson & All [18] .

3. Some induction schemes

These induction schemes will be useful during the construction and the proof process. They will be described more precisely in the appropriate section. They correspond in a sense to the induction schemata in Affirm [20] .

4. Some theorems

This part contains properties or theorems which can be deduced from the axioms. These theorems may be useful in the proofs.

Fig. 1 describes the specification of the positive integers type. We assume that the type Bool which represents the booleans is known somewhere else with all it's suitable operations and axioms. We can make the following remarks on this example :

- We define here three sorts. The zero sort is very convenient for constructing programs and for specifying this type.

- The notation int = int ,+zero is just syntactic. It is to avoid writing int , zero everywhere in the type.

- The underscore (_) shows the places of the operands with the disfix notation of operators.

- There are plenty of overloadings in this specification. For instance there are four + operators!

- When there is more than one sort in the left part of the arity of one operator, it means that it is a multioperator. For instance P has two outputs: int and zero.

- There exists also multiaxioms (a syntactic level) in the way that an axiom is repeated when there is some ambiguity in the type of operators or when there are more than one sort before the axiom.

For instance :

int : x + 0 = x means zero : x + 0 = x
int : x + 0 = x

or

bool : x < 0 - False " bool : x : zero < *0* = False
bool : x : int < 0 = False

See figure 1 next page for presentation of the type.

- There are conditional axioms like :

x egal y =* x oiv y = True

In fact a correct definition would be :

x egal y = True --> div y = True

We accepted this syntactic simplification, in order not to overload the axioms but they are all positive conditional.

- We can see with some examples how the initial term algebra is.

. 0 is in sort zero
. sO is in sort int and [s^n0|for n > 1) are in int because s is not a multioperator. These terms can be considered as representative of classes of terms in int
. psO is in zero because of the first axiom
. ppsO does not exist because p does not apply to sort zero
. sO - ssO = 0 - sO by axiom 3
= erint by axiom 4
Then it is in inter
. The term sO + (sO - sssO) does not exist !
It can be proven that there is no ambiguity for any ground term and then this term algebra is initial.

- The axiom inter : x = erint collects all the error terms into one single class without avoiding all the classical problems with the errors in abstract data type.

- It is necessary for the evaluator to do type cheeking during the evaluation process in order to choose the correct axioms or to detect terms which do not exist. In fact these terms will not be generated with correct programs. It might be possible here to add some operators which could be applied to inter to get error propagation and these terms would then exist (for instance :

+ : (int, inter) (int, inter) -> int, inter

in spite of the existing one !)

- An equation can be used iff its two sides exist and have the same type. Then, for instance, the third theorem

int : x + (y - x) = y means
If y - x ∈ int then the two sides of the equation can be substituted and this condition has to be kept during the evaluation of terms.

```
Type Int+, zero, inter
     Int = Int+, zero
Signature
   0 : → zero
  s_ : int+ → int+
  p_ : int+ → int+, zero
  _-_ : int → int+, zero, inter
  _+_ : int int → int
 _egal_ : int int → bool
  _<_ : int int → bool
  _>_ : int int → bool
 _div : int+ int+ → bool

Axioms
     int : psx = x
     int : x - 0 = x
int, inter : sx - sy = x - y
    inter : 0 - sx = erint
     int+ : x + 0 = x
     int+ : x + sy = s(x + y)
     bool : x egal y ⇒ x div y = True
     bool : x < y ⇒ x div y = x div(y - x)
     bool : x > y ⇒ x div y = False
    inter : x = erint
     bool : x egal x = True
     bool : 0 egal sx = False
     bool : sx egal 0 = False
     bool : sx egal sy = x egal y
     bool : x < 0 = False
     bool : 0 < sx = True
     bool : sx < sy = x < y
     bool : 0 > x = False
     bool : sx > 0 = True
     bool : sx > sy = x > y

Schemes
SH1 :: [x:int+ ⇒ y = px : int+] ; 0 → out
SH2 ::
  SH11 : [x:int+, y : int+ ⇒ z = x-y : int+, y] ;
         erint → SH12
             0 → out
  SH12 : [x:int+, y : int+ ⇒ x, z = y - x : int+];
         erint → SH11
             0 → out

Theorems
 int : x - y = 0 ⇒ x egal y = True ⇒  x = y
                                   ⇒  y - x = 0
 int : x + y = y + x
 int : x +(y - x) = y
bool : x div y ∧ x div z = x div(y + z) ∧ x div y
....
```

Figure 1

IV PROGRAMMING LANGUAGE

1. Syntax

Our programming language here is very simple (but powerful), which is in one sense an algorithmic specification language but it is suitable for our purpose.

Figure 2 (next page) shows the syntax of this language.

2. Semantic

The semantic is given here as a rewrite system. This is particularly useful if we want to evaluate procedures on terms in order to make proofs ! We will use some auxiliary fonctions or combinators in order to shorten the description .

$$\mathcal{M}(\text{Elprogr, Env, Cont} \to \begin{matrix}\underline{\text{return}}\ (\text{Env}) \\ \underline{\text{Except}}\ (\text{exception name, Env}) \\ \underline{\text{kill}}\end{matrix}$$

is the basic construction of the semantic. Think of it as the meaning of one element of program (which has to be considered as abstract syntax even if it is very close to the concrete syntax) in some environments and with some particular continuations. It returns return with one environment or an exception or kill.

One environment can be considered as a list of couples $[\langle x_i, v_i\rangle]$ where x_i is a variable and v_i its value in some algebra A. Env(x) = v_i means the value of x in the environment Env. It is equal to v_i if it exists $\langle x_i, v_i\rangle$ otherwise it is undefined. We will assume all the functions to be strict but we will not use this assumption because of the programming construction process.

```
le. name        means the instruction list of "le"
                associated to name
 F. outputvar   means the output variables of proce-
                dure F
 F. inputvar    means the input variables of proce-
                dure F
 F. body        means the body of F

Except ? (Op̂(value)) = True if in the algebra on
                        which the procedure is ap-
                        plied , Op̂(value) will give
                        a domain which is not of the
                        output sort.
```

The following rewrite rules give the meaning of a succession of instructions.

$$\mathcal{M}(\emptyset, \text{Env}, [i\ ;\ l_i \mid l_e].c) \to \mathcal{M}(i, \text{Env}, [l_i \mid l_e].c)$$
$$\mathcal{M}(\emptyset, \text{Env}, [\emptyset \mid l_e^i].c \to \mathcal{M}(\emptyset, \text{Env}, c)$$
$$\mathcal{M}(\emptyset, \text{Env}, \emptyset) \to \underline{\text{return}}(\text{Env})$$

$[l_i \mid l_e]$ means a continuation by a list of instructions (l_i) and a list of exception names with a list of instructions related to each name (l_e). These two rules show the semantic of raise.

$$\mathcal{M}(\underline{\text{raise}}\ \text{name, Env, } [l_i \mid l_e].c) \to \mathcal{M}(\emptyset, \text{Env}, [l_e.\text{name} \mid \emptyset].c$$
$$\mathcal{M}(\underline{\text{raise}}\ \text{name, Env, } \emptyset) \to \text{Except(name, Env)}$$

This rule is the semantic of one basic operation which has to apply directly into an algebra

$$\mathcal{M}(x := \text{Op}(v), \text{Env}, c) \to \underline{\text{if}}\ \text{except ?}\ (\hat{\text{Op}}(\text{Env}(v)))\ \underline{\text{then}}\ \mathcal{M}(\underline{\text{raise}}(\hat{\text{Op}}(\text{Env}(v))), \text{Env}, c)\ \underline{\text{else}}\ \mathcal{M}(\emptyset, [\text{Env}, \langle x, \hat{\text{Op}}(\text{Env}(v))\rangle], c)$$

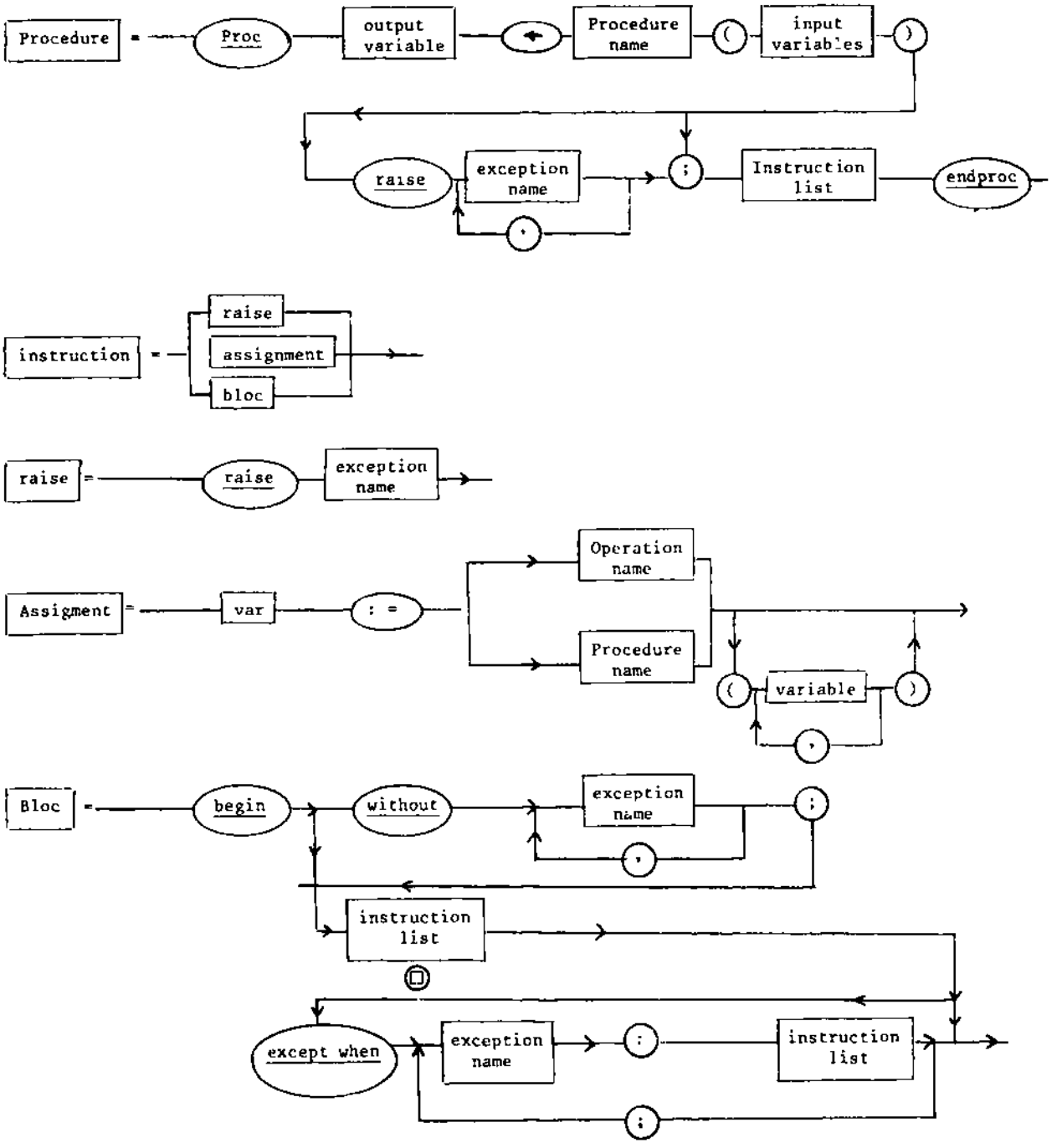


Figure 2

These three rules are for the semantic of the calling of other procedures. It uses one auxiliary function $\overrightarrow{\mathcal{F}}$.

$$\mathcal{M}(z := F(v), Env, c) \rightarrow \mathcal{M}(z \leftarrow \overrightarrow{\mathcal{F}}\ \mathcal{M}(\emptyset, [\langle F.\text{inputvar}, Env(v)\rangle], [F.\text{corps} \mid \emptyset]), F), Env, c)$$

$$\mathcal{M}(z \leftarrow \overrightarrow{\mathcal{F}}(\underline{\text{return}}\ (E), F), Env, c) \rightarrow \mathcal{M}(\emptyset, [Env, \langle z, E(F.\text{outputvar})\rangle], c)$$

$$\mathcal{M}(z \leftarrow \overrightarrow{\mathcal{F}}(\underline{\text{except}}\ (\text{name}, E), F), Env, c) \rightarrow (\underline{\text{raise}}\ \text{name}, Env, c)$$

All the rules which follow show the semantic of the bloc construction. It shows taht the begin end construct is a kind of parallel evaluation of processes.

$\mathcal{M}$(begin without nex ; lp exceptwhen le end, Env, c)
→ $\mathcal{P}_0$ ($\mathcal{P}_1$ (l_p, Env, nex), Env, [∅, l_e], c)

Here are the successions of processes.

$\mathcal{P}_1(l_i \square l_p, Env, nex) \rightarrow \mathcal{R}(\mathcal{P}_1(l_i, Env, nex), \mathcal{P}_1(l_p, Env, nex), Env, nex)$
$\mathcal{P}_1(l_i, Env, nex) \rightarrow \mathcal{R}(\mathcal{M}(\emptyset, Env, [l_i|\emptyset]), \emptyset, Env, nex)$

When a process is killed, it leaves the others to work normally.

$\mathcal{R}$(kill, P, Env, nex) → P
$\mathcal{R}$(P, kill, Env, nex) → $\mathcal{R}$(P, ∅, Env, nex)

If a process terminates normally it tries to export its results after killing the others.

$\mathcal{R}$(return (E), P, Env, nex) → return (E)
$\mathcal{R}$(P, return (E), Env, nex) → return (E)

If a process raises one exception and if this exception belongs to the without part of the begin end it is like a killing of the process, else it is like a normal terminating one.

$\mathcal{R}$(except (name, E), P, Env, nex)
→ if name ∈ nex then P else except (name, E)
$\mathcal{R}$(P, except (name, E), Env, nex)
→ if name ∈ nex then $\mathcal{R}$(P, ∅, Env, nex) else except (name, E)

These three rules give the final result of the begin end construct following the process which succeeded.

$\mathcal{P}_o$(kill, Env, c) → $\mathcal{M}$(∅, Env, c)
$\mathcal{P}_o$(return(E), Env, c) → $\mathcal{M}$(∅, Env, E], c)
$\mathcal{P}_o$(except(name, E), Env, c) → $\mathcal{M}$(raise name, [Env, E], c)

This langage is in fact too rich and we can prove that the extension of homomorphism h (between T_Σ and any algebra of Alg_Σ) cannot be done for every program. We will restrict our programs such that the definition of programs plus the definition of the semantic plus the definition of abstract types is a canonical system. This means terminating programs and no real nondeterminism. The schemes which belong to the astract types will help us in doing that.

Not that this eliminates programs such that

```
Proc y ← Anynumber (x: int)
     begin
          y : = x □ y : = Anynumber (s(x))
     end
endproc
```

which is unbounded determinism.

In order to help the proof process we will also suppose that we have unique assignment programs (we can then replace the : = by equalities during the proofs). The responsability to obey all these restrictions is left to the programmer because it is highly non-decidable for all programs.

V PROGRAM CONSTRUCTION PROCESS

One program at the begining can be considered as a name, a set of input variables and their types, an output variable and its type and a list of exceptions which can be raised by the program.
Let y:t'←F(x:t)raise exl ; such an expression where to simplify we use only one input variable and one exception name.

The program construction process consists of choosing a set of input variables and to use one of the schemes which belong to the description of the corresponding types.

This transformation can be expressed as in figure 3 where :

. x, x', j... are only one variable but this scheme can be generalized to more than one as in the example.

. The $inex^i_j$ mean the exceptions or the sorts which can be obtained using multioperators θp_i or $\theta p'_i$ such that the subscheme SH^i_j can be evaluated normally.

. The $outex^i_k$ mean the exceptions or the sorts such that no subscheme SH^i_j can be used without giving a sort outside the normal output of the operators.

. We assume that for each scheme SH on x there exists one function V : T → ℕ such that :
∀x ∈ T V(x') < V(x) on each subscheme SHI_i.

That means that the scheme corresponds to a well-defined or dering on x and is particullarly useful for making proofs.

. In the program generated :
- $SP_i ... denotes subproblems which have to be expressed directly by the user or as new subproblems to be solved in the same way.
- erF is one exception which is systematically added to each procedure. It can be simplified if it can be proven at the end that it is useless.
- We assume here that we can use recursion on F(t : = F(x')) but this can be easily avoided. If the recursion is used then an erF exception with its corresponding subproblem $SPerF is generated.
- An exception for each subproblem $SP_i (ie er$SP_i) and a new subproblem are generated.
- All this is immediately and automatically generated simply by pointing to a scheme. If some of the subproblems seem to remain unsolved or not easily expressed, the programmer can choose another scheme and so on...

Example

We will construct the well known gcd program which can be considered as a toy problem, but which is more difficult that most of the practical real problems in the concepts it invokes. The heading is:
Proc p:int⁺ ← gcd(x:int⁺, y:int⁺)
We will choose to use the scheme SH2 on type int⁺. This leads automatically to :

```
Proc p:int+ ← gcd(x:int+,y:int+)raise ergcd ;
     begin without erint ;
          z : = x-y ; P1 : = gcd(z,y) ; $SP1
        □ z : = y-x ; P1 : = gcd(x,z) ; $SP2
Except when
       0 : $SP3
   ergcd : $SP4
 er$SP1 : $SP5
 er$SP2 : $SP6
 end  endproc
```

We have to "manually" build the subproblems $SP_i.
It is easy to guess that
  $SP_1 is P:P_1
  $SP_2 is P:P_1
which cancel er$SP_1 and er$SP_2 then $SP_5 and $SP_6.
  $SP_3 is P:=x
  $SP_4 by raise ergcd

An easy transformation leads then to the final program:

```
Proc p:int+←gcd(x:int+, y:int+)
  begin without erint;
    z:=x-y; p:gcd(z,y)
  □ z:=y-x; p:=gcd(x,z)
  except when
    0:p:=x
  end
endproc
```

which is quite elegant and well-structured.

```
SH ::
 SHI_1 :[x : T ⇒ x' = θ p_1(x) : T ;
        z : = θ p'_1(x) : T']
        {inex^1_j → SH^1_j}
        {outex^1_k → out}
 SHI_2 :[x : T ⇒ x' = θ p_2(x) : T ;
        z : = θ p'_2(x) : T"]
        {inex^2_j → SH^2_j}
        {outex^2_k → out}
 ..... ⇓
Proc y ← F(x : T) raise name,...,erF ;
  begin without ⋃ inex^i_j ;
        x' : = θ p_1(x)    ; z : = θ p'_1(x) ;
        t : = F(x') ; $SP_1
      □ x' : = θ p_2(x) ; z : = θ p'_2(x) ;
        t : = F(x') ; $SP_2
      □ ...
  except when
        outex^i_k : $SPE^i_k
        .......
        er$SP_1 : $SPerSP_1
        .......
        erF     : $SPerF
        name    : $ name
  end   .......
endproc
```

Figure 3

VI PROOF

As there is nothing really special in the proof process, we will only describe it on the example in order to conclude.

We will assume for instance a "non executable" algebraic specification of gcd is given by

- gcd(x,y) div x ∧ gcd(x, y) div y
- z div x ∧ z div y ⇒ ¬ (gcd(x, y) < z)

1. Let us prove a first lema

(1) gcd(x, y) = gcd(y, x)

We will use the induction following the scheme.

a) First subscheme

gcd(x, y) = gcd(x - y, y) then x - y ∈ int+
= gcd(y, x - y) by induction
= gcd(y, x) by second subscheme.

b) Second subscheme

idem.

c) Exception

gcd(x, y) = x by exception 0 then x - y = 0
= y by theorem 1 in int+
= g(y, x) by exception on the scheme.
□

2. Let us now prove the first part of the specification
gcd(x, g) div x ∧ gcd(x,y) div y

Will use the induction on the scheme in the same way :

a) First subscheme

gcd(x, y) = gcd(x - y, y) = gcd(y, x - y)by lemma 1
= gcd(y, x - y) div y ∧ gcd(y, x - y) div x - y by induction
= gcd(y, x - y) div y + (x - y) ∧ gcd(y, x - y)div y by theorem of int+
= gcd(x, y) div x ∧ gcd(x,y) div y by subscheme 1 and theorem 3 of int+.

b) Second subscheme

Idem

c) Exception

gcd(x, y) = x then x - y = 0 and x = y
gcd(x, y) div x ∧ gcd(x, y) div y
= s div x ∧ y div y which is true by property of div.

3. The second part of the specification is now to prove
z div x ∧ z div y ⇒ □ (gcd(x, y) < z)
which can be proven using the same kind of induction.

This example shows that the proof is not very easy, even for such a simple example because of the theorems or lemmas which have to be used. In our opinion, this kind of proof cannot be done automatically by a present theorem prover (with the discovery of lemmas).A nice proof checher would be preferable.

V CONCLUSION

This method, this toy example and the simple proof do not intend to describe all the tools which have to be in such a system, l.'e claim here that when we try to be very precise (and we have to when we build correct programs) all the concepts which belong to this paper have to be contained in one way or anotlier in the system, which leads to many difficult theoretical problems not completely solved at this time.

What we can hope for in the near future is the effective implementation of such partial systems which will become more and more powerful, coupled with meaningful research on abstract data type theory, Specification languages, Theorem provers or proof cheekers and rule rewriting systems.

"ACKNOWLEDGMENTS ."

The authors wishes to thank M. Bidoit, M.C. M. C. Gaudel, C. Gresse, S. Kaplan for many exciting discussions on this subject.

REFERENCES

[1 J Manna,Z. & R. Waldinger, "Deductive synthesis of the unification Algorithm" In Computer Program synthesis methodologies. A. Biermann & G. Guiho Editors. D. Reidel'Publishing Co 1983.

[2] Bidoit, M.& C. Gresse & G. Guiho, "A system which synthesiezs array manipulating programs from specifications. Proc 6th 1JCAI-79. Tokyo pp. 63-65.

[3]Kodratoff, Y, "A class of functions synthesized from a finite number of examples and a LISP program shceme". International J. of Comp. and Inf. Sciences 8, 1979, pp. 489-521.

[4] Jouannaud,J. P. & G. Guiho, "SISP/I An interantive system able to synthesize functions form examples". Proc 5th IJCAI. M.I.T., August, 1977.

[5] Biermann, A. W. & D. R. Smith, "The hierarchical synthesis of LISP scanning programs". Information processing 77, North Holland, 1977, pp. 41-45.

[6]Moriconi, M. "A designer/verifier's Assistant" IEEE Transactions on Software Engineering, Vol Se-5 N° 4, July, 1979.

[/]Barstow, D. R. "Automatic Construction of Algorithm and Data Structures using a Knowledge base of Programming Rules". PHD Dissertation, Stanford Memo A1M-308, 1977.

[8]Good D. & All, "Report on the language Gypsy" Version 2. 0. Institute for computing Science and computer applications. The University of Texas. Austin Texas 1978.

[9]Shostak, R. E. and R. Schwartz, Mel 1iard-Smith P.M. STP : ?A Mechanized Logic for specification and verification". 6th conference on Automated Deduction, NY 1982.

[I0]Huet, G. "Projet Formel", INRIA, France 1983.

[II]Bouge, L. "Modelisation de la notion de test et de programmes". These 3e cycle LITP Paris Novembre 1982

[12]Thatcher, J. W. and E. G. Wagner and J. B. Wright, "Specification of Abstract Types using conditional axioms , IBM report 6214, September 1976.

[13]Coguen, J. A. and J. W. Thatcher and E. G. Wagner, "An Initial Algebra approach to the specification, correctness and Implementation of abstract data type. IBM report KC 6487, October, 1976.

[I4]Goguen, J. "Order Sorted Algebras : Exceptions and Error Sorts, coercion and Overloaded operators. Report IT 14, S.R.I. December, 1978.

[15]Wirsing, M. and M. Broy, "Abstract data types as lattices of finitely generated Models". 9th MFCS ,Rydjyna, September, 1980.

[I6]Bidoit, M; "Algebraic Data Types. Structured Specifications and Fair Presentations". Colloque AFCET/MA ,Paris, March, 1982.

[17]Kaplan, S. "Un langage de specification de types abstraits algebriques". These 3e cycle LRI- Orsay, Fevrier, 1983.

[18]Boisson, F. and G. Guiho and D. Pavot, "Algebres a Operateurs Multitypes". Rapport interne LRI-Orsay, Mai, 1983.

[19]Gresse, C. "Automatic programming from Data Type decomposition paterns". 8th IJCAI-83, Karlsruhe, August, 1983.

[20]Loeckx, J. "Proving Properties of algorithmic Specifications of Abstract data Types in AFFIRM". Memo-29-JL U.S.C July 1980.