Formal Definition in Program Development

C. B. Jones, IBM Laboratory Vienna.

ABSTRACT

The intent of the current paper is to show how a large problem like compiler development can be divided in a way which provides a structure for arguments of correctness. Although mechanically checked proofs are not envisaged, the use of formal notation is recommended so that the basis for correctness arguments exists.

The paper reviews three topics; the first two are relevant particularly to the development of compilers; the third is more general. The subject of the first section is the style of language definition to be used as a basis for development. Beginning with a small language, possible ways of describing added features are discussed.

The selection criterion for definition techniques is their usability in developing a specification of the compiling process: it is this development which is the subject of the second section.

The third section briefly reviews the process of "Formal Development" which has been described more fully elsewhere.

0. INTRODUCTION

This paper provides an overview of a number of pieces of work related to program correctness. Since the possibility of having completely mechanically checked proofs for large programs appears to be some way off, the approach taken is one of documenting a "justification" for human readers. The paper will attempt to show how a large problem can be decomposed into small enough steps that such justifications can be convincing.

The particular problem to be considered is that of developing a compiler. Of course, a compiler is a very special program, but to oversimplify for a moment one can consider that it is special only in requiring two extra stages of development to precede that which is applicable to any program.

Three major parts of the compiler development problem are discussed in sections 1 to 3 of this paper. The first section discusses the definition of the language to be compiled. This definition of the semantics of the source language provides the overall correctness criteria for the compiler: whatever results can be deduced about a program written in the source language must also be true when running the object code produced by compiling that program. The discussion identifies important properties of a language definition to be used in the next stage.

Given a particular object machine, the next step is to develop a mapping from the source to the object language. How this is done is the subject of the second section of the paper. Examples are given of mapping the abstract state objects of the definition onto the store of a target machine (cf. refs. [9, 15]).

Any top-down development process must begin with an overall correctness criterion: that is, a specification of an input/output relation. The purpose of section 2 was to show how exactly this can be provided for the compiling problem. The process of developing such a specification by a step-wise

process to a running program is discussed in section 3. It is argued that the use of data abstraction and appropriate choices of implicitly defined functions can provide the structure for justifications of large programs.

Since no specific length limit was given to the author, it must be confessed that his lack of time is the reason that all three sections are not written up fully. The ideas behind section 3 are well enough documented in other papers that an overview should suffice. Although the general direction to be followed in the work covered by section 2 is clear, the example provided is very small. This runs into the usual problem that in such cases it is easy to "see" the correctness, whereas it is precisely the inability of our "small head" to contain a large problem that gives rise to the need for a justification. Only section 1 approaches the level of completeness the author would have liked to attain.

The process of "Formal Development" outlined in section 3 is applicable to any programming problem. As was observed above, it is an oversimplification to think that it is therefore sufficient to show how to tackle any other computing task. In precisely the way that for a compiler it was a significant problem, generating the input/output relation for other tasks will be difficult. It is abnormal for initial specifications to be couched in terms of such relations, and, other than arguing that its production should be the first step, the current paper offers no help as to how it can be obtained.

The emphasis throughout is on the <u>method</u> of decomposing a large problem into small enough steps to provide a justification. Certain common requirements result from this, one of which is the necessity to use a formal notation: only then will it be possible to document justifications. It is not, however, the intention to argue for one particular <u>notation</u>.

A further technique, which becomes almost a necessity if proofs are to be of an acceptable length, is the use of abstraction. Dijkstra has used the terms "Operational Abstraction" to cover implicitly defined operations and "Representational

Abstraction" to refer to the postponement of unnecessary data properties. Both of these techniques will be used, but again no particular notation is recommended.

It is the intention in the current paper to concentrate on techniques rather than deep results. Transcending the choice of a particular mathematical discipline to underly the work are the practical steps which must appear in any justification: the attempt is to throw some light on these.

1. LANGUAGE DEFINITION STYLE

This part of the paper suggests certain properties of a language definition which will facilitate its subsequent use in development of a translator design. Notice that a definition constructed along the proposed lines will not necessarily suit other purposes like proving programs correct in the defined language.

Although easy to express (see summary) it is not always easy, given a language feature wanting definition, to find a formulation with the required properties. For this reason the plan adopted below is to consider separate language features and possible formulations for their definitions. An attempt has been made to deliberately separate the language concepts. In this way it is hoped that the reader can see the requirement for the different formulations in isolation, whereas in a complete definition of a language it is often difficult to see the source of the complexity. If the current approach were executed on all of the features of the respective languages one would then be able to explain ref. [1] and ref. [4] as the "cross products" of their respective formulations. For, although the notation of the latter is a step forward, both definitions possess the properties discussed.

Rather than discussing notation, the emphasis below is on finding the appropriate model for a language feature. The

distinction between and similarities of the so-called "operational" and "mathematical" approaches are considered but it is argued that there are criteria for choosing the model which transcend the distinction.

Most of the solutions discussed are behind a number of current language definitions. Only the solution of the goto problem is novel.

In order to provide an overall context for the decisions made below it is worth pointing out the origin of the difficulties which led to a reconsideration of some aspects of the "VDL" (Vienna Definition Language) notation. During 1969/70 the author had the pleasure of co-operating with P. Lucas and his colleagues on attempts to document correctness arguments for compiling algorithms based on the then current formal definition of PL/I (ref. [24]). Although the feasibility was shown in refs. [14, 7, 9], a number of difficulties were encountered. With the exception of the use of the control mechanism (see 1.4 below) these were certainly not caused by any shortcomings in the formal notation itself. The common origin of most problems was, in fact, the tendency to be "over specific" in its use. By this is meant that the abstract interpreter used to define the language sometimes indicated results not required by the language. Although it was certainly possible to deduce that such results had no effect on the final outcome, this proof frequently went far beyond the part of the language under consideration.

A simple example was the use in some VDL models of a never recurring "unique name generator" to obtain new locations when required. This certainly gave a sufficient model which gave the required outcome. However, if one wanted to prove correct a stack implementation, in which the locations of previously closed blocks could be re-used, one was more interested in the necessary conditions of uniqueness. Now it was, of course, possible to prove that, since new locations were initialised to an undefined state, it is permissible to re-use discarded locations. However, one was paying with a very expensive proof the saving of relatively few lines of definition.

The basic maxim to be followed then will be to avoid giving properties to the definition which are not required by the language.

Before coming to the language features proper, a brief section on notation is offered.

1.1 Notation

The formulae which follow should offer no difficulty as regards their use of notation for set and logical operations. Use is also made of simple programming constructs like "if then else" and conditional statements. This section confines itself to sketching meanings for the non-standard items. For a more complete discussion see part 1 of ref. [4].

The concept of Abstract Syntax, introduced in ref. [17], was used to advantage in the VDL definitions. Not only is its use still considered mandatory to divorce a semantic definition of a language from the richness of its syntactic alternatives; but also the subsequent parts of this paper will show its importance in translator development.

The abstract syntax notation now used has been changed somewhat from ref. [24] in order to shorten and clarify descriptions. Given a grammar it can be read as set equations as follows -

Elementary Objects as names of unit sets -

ABC ~ {ABC}

Non-terminals as names of the sets defined in the respective rule.

Rules for defining alternatives of non-terminals -

W = X|Z ~ W = X ∪ Z

Rules for introducing constructors -

X :: YZ ~ X = {mk-X(y,z) | y∈Y ∧ z∈Z}

Furthermore, selectors can be introduced -

X :: s-n1:Y s-n2:Z ~ s-n1(mk-X(y,z)) = y
s-n2(mk-X(y,z)) = z

The use of a nonterminal name which ends in "*" ("-set") denotes a list (set) of objects of the class defined by the name without its suffix.

To test for membership of a set -

is-W(o) or is-X(o)

Other than decomposition by selection it is possible to decompose by using the constructor on the left of a definition-

let mk-X(y,z) = x ~ let y = s-n1(x)
let z = s-n2(x)

Use is also made of this binding of names in a cases construct-

cases w:
mk-X(y,z) -> f(y,z) ~ is-X(w) -> (let mk-X(y,z) = w;
f(y,z))
mk... is...

At the points where it is necessary to discuss more carefully the manipulation of functions, use is made of the Lambda notation-

f(x) = ...x... ~ f = λx. ...x...

However, these uses will often be "sugared" in Landin's style (see ref. [13])

let x = e;)
) ~ (λx.g(x))(e)
g(x))

Maps are used where the graph of a function can be explicitly constructed -

[d1 -> r1]	explicit definition
[d -> r | p(d,r)]	implicit definition
+	joining

are the counterparts of the set concepts.

To come now to the problem of Semantic Definition. The definitions given below will be written in terms of functions from stated domains to ranges. In using the term functional semantics it is intended to emphasise the distinction from the VDL style models, ref. [16], in which an explicit control component exists in the interpreting machine (this point is discussed more fully below in connection with the goto construct). The relation between functional semantics and mathematical semantics ref. [22] is of more interest and will be reviewed in connection with several of the language features considered. To begin with it is worth showing the definition of a language which is itself functional and thus affords an easy path to functional semantics.

1.2 Expression Language

Consider the language given by the following abstract syntax rules -

B1 expr = inf-expr | var-ref | const

B2 inf-expr :: expr op expr

B3 var-ref :: id

B4 const :: INTG

The class op is not further specified other than by the existence of a function -

B5 apply-op : INTG op INTG -> INTG

Now, for a given set of denotations (the term "environment" is avoided because it will be used below) for the free identifiers-

B6 DEN : id -> INTG

the denotation of an expression, which is also an integer, is given by -

```
B7 eval-expr(e,den) =
      cases e:
      mk-inf-expr(e1,op,e2) ->
         (let v1=eval-expr(e1,den);
          let v2=eval-expr(e2,den);
          result is (apply-op(v1,op,v2)))
      mk-var-ref(id) -> den(id)
      mk-const(n) -> n

   type: expr DEN -> INTG
```

This definition has the property that the denotation of a composite expression depends only on (and can therefore be constructed from) the denotations of its component expressions.

The introduction of the concept of a dynamic assignment to a variable is perhaps the most distinctive feature of a programming language. The fact that, in contrast to mathematics, one is forced to consider the value of a variable at a given point in time poses problems for the definition of semantics.

1.3 Assignment Language

Consider the language whose programs consist of a sequence of assignment statements (as-st) which can be described -

C1 as-st*

C2 as-st :: s-lhs:id s-rhs:expr

The effect of such a sequence of statements will be to transform some initial set of denotations for the variables, step by step, into their final denotations. Thus it is no longer sufficient to consider the DEN as an argument to the interpretation: the DEN required as the argument to the second (and subsequent) calls of the interpretation may have been changed by the interpretation of the first assignment. The function given below appears to ignore this problem by simply omitting all mention of the DEN. This is done because the intention is to offer a number of different explanations. It should be possible to see the intent of what is written if one accepts that assign "changes" the DEN for the given id, and assumes eval-expr uses the current DEN -

C3 int-st-l(st-l,1)

```
C4 int-st-l(st-l,i):
      if i ≤ lst-l
         then
            (let mk-as-st(lhs,rhs) = st-l[i];
             let v:eval-expr(rhs);
             assign(lhs,v);
             int-st-l(st-l,i+1))
         else
            I
```

type: as-st* INTG =>

It is possible to consider three ways of reading such formulae. The first possibility is to read them as programs.

As such, each function would correspond to a subroutine which refers to one non-local variable (i.e. the DEN). It is, of course, the same non-local variable referenced by the modified eval-expr and all calls of int-st-1. The sub-program assign is trivially defined to modify this variable. The separator ";" has its usual ordering implications. Subroutine type clauses are written with "=>", and their calls are marked with a ":", in both cases to distinguish them from pure functions.

Given this view and using the notation of ref. [12], the types could be given in full by -

```
int-st-1  :: DEN:as-st* INTG ->
eval-expr :: DEN:expr -> INTG
assign    :: DEN:id INTG ->
```

With this simple, constructive, view it is already possible to discuss one of the desirable properties of a language definition. McCarthy has used the term "Small State" to describe a definition in which only those things which change very dynamically are put into the state used by the defining functions. This is in contrast to a "Grand State" style in which all variables, with the possible exception of program text, are put into the state. Although in this trivial language there is no incentive to do so, it would have been possible to make the statement counter part of the state and show sequencing by a side effect on this new non-local variable. The disadvantage of taking this approach is that it would not be clear, without further inspection, that the statement counter could not be affected by, for example, eval-expr.

Returning now to the discussion of alternative views of the function int-st-1. The second possible interpretation is to regard all functions as taking an extra argument and yielding an extra result: in each case a DEN. This, which is the view of ref. [1], would give-

```
int-st-1:  as-st* INTG DEN -> DEN
eval-expr: expr DEN -> INTG DEN
assign:    id INTG DEN -> DEN
```

(In fact it is often possible to simplify; in the above example, since it can cause no changes, eval-expr need not return a DEN.)

But it is no longer possible to rely on the programming view of ";". It is necessary to describe it as a combinator between functions. The task of doing this is complicated by the various alternative contexts and it is easier to show the result which would come from using the combinator -

```
int-st-l(st-l,i,den) =
      if i ≤ lst-l
         then
            (let mk-as-st(lhs,rhs) = st-l[i];
             let (v,den1) = eval-expr(rhs,den);
             let den2 = assign(lhs,v,den1);
             result is (int-st-l(st-l,i+1,den2)))
         else
            den

type: as-st* INTG DEN -> DEN
```

Since the "lets" are now on pure functions, they are simply a sugared form of lambda expressions.

The third view one could take of the function int-st-l is that of ref. [22]. The comment was made on the definition of the functional language, that the denotation of its sub-components was all that was necessary to determine the denotation of a unit. By regarding the denotation of an assignment statement as a function it is again possible to enjoy this property. (That this is so is more clearly shown if the abstract syntax of a composite statement is given recursively.) The resulting types would be-

```
int-st-l:  as-st* INTG -> (DEN -> DEN)
eval-expr: expr -> (DEN -> INTG DEN)
assign:    id -> (INTG -> (DEN -> DEN))
```

Again in this view it is necessary to define ";" as a combinator. But now the fact that the units to be combined (after applying the functions to the static components) are basically functions of the type DEN -> DEN means that the very pleasing model of functional composition is adequate.

The Oxford group (refs. [22, 23, 19]) have gone rather far in designing combinators which would permit formulations like -

```
int-st-1(st-1,i) =

      λden.(if i ≤ lst-1
               then
                  (let mk-as-st(lhs,rhs) = st-1[i];
                   int-st-1(st-1,i+1) °
                   COMB(eval-expr(rhs),
                        assign(lhs)))
               else
                  I)

type: as-st* INTG -> (DEN -> DEN)
```

where -

```
    COMB(vf,uf) =
       λden.(λv,den1.(uf(v)(den1))(vf(den)))
```

(It should be made clear that if this had been written by a genuine devotee of mathematical semantics, it would have looked very different. It is the current author's view that excessive zeal in shortening definitions makes them less rather than more readable, cf. ref. [19], ref. [1]).

Since this is a function one can look at its result for a test program! Consider -

```
p = (x := 1;
     y := x + 2)
```

Then after reduction-

```
int-st-1(p,1) =
    (λden.den + [y -> den(x)+2]) ° (λden.den + [x -> 1])
```

which is the result expected. (This exercise is somewhat more illuminating on larger examples.)

(Reverting for a moment to the earlier discussion of grand versus small state approaches, it is worth noting that it would have been possible to make the (undesirable) step of putting the statement counter in the state and still give a definition in terms of a function to functions from states to states. It would not, however, have been possible to provide such simple combinators. In particular, the static role of the statement counter is required to provide the required decomposition of the semantics.)

If the appropriate combinator definitions were written, we have now provided three ways of reading the formula int-st-1. The question of which should be used must now be discussed.

The position advanced in the next part of this paper is that the interpretive view is useful during the development of the translator mapping and it is only then that one need take the view of mapping source programs to functions. Observe, however, that thinking in terms of combinators required that certain problems be resolved more carefully: this leads to retention of also this view of the formulae. Thus the notation will develop the style above; definitions written in this style will be manipulated as if they were operational; the mathematical view may be appealed to when decisions are otherwise unclear.

With the amount of notation introduced so far it would be easy to define "if then else" or any variant of the "for". In doing this for any reasonably complex language the question arises as to how much notation is it reasonable to add to the meta-language. Would it, for instance, be acceptable to have a while construct? The answer must always be sought in the ease of reasoning about a construct. Thus in ref. [4] both while

and for constructs have been included, but they are of a much more restricted kind than the FOR of PL/I which was being defined.

This topic brings us to the challenge of giving a semantic definition of goto.

1.4 Goto Language

The long debate on the morality of the goto construct has not yet resulted in its banishment from descriptions of languages by standards committees. To be serious, it appears to be valuable to have some mechanism for abnormal sequencing situations and an ability to provide formal definitions for them may be one of the tools for comparison.

The problem with defining goto is that, other than the local hop, its ability to leave or enter phrase structures upsets the attempt to state the semantics of a unit solely in terms of the semantics of its sub-units. In this section the compound statement is chosen as the phrase structure whose semantics should be provided. Although this is simpler than block exits, it has the property that the same phrase structure can be terminated and initiated abnormally so that with a shorter definition both problems can be discussed. The subject of block terminations is discussed in the next section.

The technique for modelling goto employed in ref. [24] was to introduce a control stack component into the state of the abstract machine. (In fact the control was more general, but this point is discussed below in connection with arbitrary evaluation order.) Instead of describing the meta-language directly as functions, a VDL definition is itself described by an interpreting function (called LAMBDA in ref. [24]). A step of the interpreting function removes the top instruction from the control stack: if this operation is elementary, it is obeyed and the next step of operation is performed; in the case

of a "macro" operation, the appropriate operations are put on the control stack so that the next step will encounter them.

So far this can be thought of as a way of describing functional application. The purpose, however, of making the control component an explicit part of the state was to make possible its explicit manipulation. Thus one way to model goto out of a phrase structure was to define the "obvious" operations structured in line with the phrase structure, but to simply delete from the control component any operations which corresponded to parts of the program being jumped over.

The effect of this was that, in general, it was not possible to present arguments whose inductive structure followed the phrase structure of the program. It was of course possible to present proofs, but they had to be by induction over the sequence of states generated by LAMBDA. One could argue that this is precisely the undesirable effect of the goto, but in fact the definition had gone too far. It was one of the places where the generality, in this case to change the control in any instruction, forced one to show that, in precisely the places one did not require the power, it was not used.

In fact the deletion of parts of the control was sometimes only used for exits from blocks: the reason that it was not used for more local phrase structures was the solution adopted for handling abnormal entry into such phrase structures. Basically, some definitions adopted a "current statement selector" which was simply changed to point to the next statement. This made goto into and out of phrase structures very easy to describe providing there was no special epilogue action to be performed on exit from the phrase structure. (This can, in fact be viewed as absorbing part of the LAMBDA function into the definition). However, such definitions tended to cloud the normal action by the necessity to describe finding the successor to an embedded unit by manipulating the pointer (see, for example, the treatment of END in ref. [25]).

The current author became convinced that setting up the normal action and letting a goto "take the machine by surprise" was

the wrong model. The proposal made was that any unit which could result in abnormal termination should return an extra "abnormal" result, which was some null value in the normal case. Any call of a function which could result in an abnormal return must test for this possibility and perform appropriate actions. (Together with W. Henhapl, who provided the statement selector treatment, this was written up in ref. [6]).

In order to define Algol 60, in which it is possible to goto into both "if" and compound statements, it was necessary to address the other part of the problem. The approach employed in ref. [1] is to provide functions which run through the phrase structure without executing but setting up all of the actions to follow. Since these functions prompted the execution as to where to begin they became called "cue-functions" (as in acting - Das Stichwort).

Consider, for example, the following -

```
goto l;
   if p
      then l:s1
      else s2;
   s3
```

Not only should this, rather odd, transfer of control get to s1 without evaluating p, it should also set up the events to be performed thereafter so that s2 is skipped and s3 is next considered.

The completely functional definition of Algol given in ref. [1] became tedious because of the many places where the effect of a goto can cause a change of events and therefore the abnormal return value must be tested. It was, however, clear that the most common action was simply to refrain from executing the next operation and pass back the abnormal value to the next level. In fact there are very few places where it is necessary to describe any special action. Based on this observation P. Lucas proposed that adopting some unit to trap the

interpretation where the action was required would leave one free to drop the "test and return" case by convention.

This abbreviation is the one used in ref. [4] and below. All normal returns are written omitting the (implied) return of a nil value for ABN. Non-nil values for ABN are returned explicitly by the exit statement. Normal action on being returned a non-nil ABN value is to terminate also the calling function abnormally with the same value for ABN. An explicit action to be performed for a non-nil ABN value is defined by means of a trap exit unit bracketed together with the statement to which it applies. Completion of the trap unit completes the containing function.

The development of this idea has been described in terms of an interpreter partly because this is how it actually occured and partly because it is probably easier to first read the following functions in this way.

(In fact the separation of the, largely similar, functions int-ns-1 and cue-int-ns-1 would probably not be made if one were taking the purely interpretive view: it is only for the more mathematical view given below that the functions are written separately.)

The language considered is given by the following abstract syntax. It is assumed that among the unlisted statement types is something like the assignment of the previous section which would force retention of the DEN component.

```
D1 st = goto-st | cpd-st | ...

D2 goto-st :: id

D3 cpd-st :: nmd-st*

D4 nmd-st :: s-nm:[id] s-body:st

   id not further defined
```

The defining functions can now be given (the "∪" operator means "the unique object satisfying") -

```
D5 int-st(st):
      cases st:
      mk-goto-st(lab) -> exit(lab)
      mk-cpd-st(ns-l) -> int-ns-l(ns-l,1)
      .
      .
      .

   type: st =>


D6 int-ns-l(ns-l,i):
      if i ≤ lns-l
         then
            ((trap exit (lab) with
               if is-contained(lab,ns-l)
                  then cue-int-ns-l(ns-l,lab)
                  else exit(lab);
             int-st(s-body(ns-l[i ])));
            int-ns-l(ns-l,i+1))
         else
            I

   type: nmd-st* INTG =>
```

```
D7 cue-int-ns-l(ns-l,lab):
      let i = (ιi)(is-contained(lab,<ns-l[i]>))
      if lab = s-nm(ns-l[i])
         then int-ns-l(ns-l,i)
         else
            (trap exit(lab) with
               if is-contained(lab,ns-l)
                  then cue-int-ns-l(ns-l,lab)
                  else exit(lab);
             cue-int-ns-l(s-body(ns-l[i]),lab));
            int-ns-l(ns-l,i+1))

   type: nmd-st* id =>


D8 is-contained(lab,ns-l) =
      (∃i)(s-nm(ns-l[i]) = lab) ∨
      (∃j)(is-cpd-st(s-body(ns-l[j])) ∧
               is-contained(lab,s-body(ns-l[j])))

   type: id nmd-st* -> {true,false}
```

It was observed above that a deeper understanding is often obtained by viewing a meta-language construct in mathematical semantics terms. This view is now attempted of the above constructs. First it is necessary to uncover what has been hidden in the "=>" of the type clauses -

```
   int-st:        st -> (DEN -> DEN ABN)
   int-ns-l:      nmd-st* INTG -> (DEN -> DEN ABN)
   cue-int-ns-l: nmd-st* id (DEN -> DEN ABN)
```

It is easy to define the denotation of two meta-language statements separated by ";" in terms of their individual denotations.

```
   st1;st2  ≃  λden.(let (den1,abn1) = st1(den);
                     if abn1 = nil
                        then st2(den1)
                        else (den1,abn1)
```

This gives us the way of creating a function whose type is (ξ -> ξ ABN) from two functions of similar type: the fact that the test is dynamic is unavoidable because the occurence of the goto will, in general, depend on the state.

It is also possible to write a very straightforward combinator for the trap exit but, if this results in an equally dynamic action, the fact that the trap exit body again uses defining functions applied to the whole text (of the current unit) would make it impossible to ascribe a semantics of the required type to a unit. The key observation is that although the labels which will come to the trap are unknown (in the sense they may be either contained or free within the unit), the set of labels for which one can do something is known: it is precisely the set of contained labels.

This point can be illustrated by the following example -

```
S =    if p then goto l
           else goto m;
     l:S2 /*no contained labels */
```

Then -

```
int-ns-l(S,1) =
   let (den1,abn1) = int-st(if p then goto l else goto m);
   (abn1 = nil -> int-st(s2)
    abn1 = 1   -> cue-int-ns-l(S,1)
    T          -> exit(abn1))
```

Now since -

```
cue-int-ns-l(S,1) = int-st(S2)
```

it can be seen how to construct the denotation of S. This was, of course, a trivial case. But even where the graph of goto statements introduces looping, an equation will be given whose fixed point can be sought.

It should be conceded at this point that it is also possible to write "definitions" using exits which do not permit static combination. This is a cause for further consideration.

The above mechanism is not the one usually employed in giving mathematical semantics: the mechanism which appears to have been accepted (cf. ref. [23]) is that of "Continuations". Basically, the denotation of a label is the function (say ξ -> ξ) which represents starting at that label and running to the end of the program! This is certainly a more powerful concept: that it can define more general languages, the current author found out to his cost when he tried to show that it was possible to eliminate continuations in ref. [21]. However, the maxim is to be sparing on power in the meta-language and using continuations to model Algol 60 labels (ref. [19]) may be too general. It seems unfortunate, for instance, that in -

```
while p do
   begin
      if q then goto 1;
      S1;
      1: S2
   end
```

the label 1 cannot be "treated locally".

The actual choice between continuations and the model offered here must be made in the context of the use of the language definition. Since the Oxford group has an interest in proving compilers correct it will await a larger example before this can be decided. The experience with basing correctness arguments on exit is, so far, encouraging.

1.5 Block Structured Language

Both blocks and procedures permit their user to introduce a local level of naming. Since the names defined within different (even nested) blocks do not have to be distinct, the simple DEN of 1.3 will not suffice as a state. Consider the case of a language in which no recursion is allowed. It is necessary to "remember" the value of a variable, say x, over the lifetime of a nested block in which another variable x is declared. One way to overcome this problem would be to make all identifiers statically distinct by, for example, qualifying them with a unique block number.

The static renaming scheme would not, however, be adequate if recursion were also allowed. It would then be necessary to keep distinct, multiple instances of a variable which is declared in a recursive block.

Before considering the passing of procedures as parameters, it is appropriate to discuss call-by-reference since its solution introduces a tool which makes the remaining problems both easier to state and solve.

Consider the following -

```
begin
   int a;
   proc p(x); int x; x := a;
   p(a)
end
```

If the variable a is passed by reference, the parameter x will refer to a. In an implementation the non-local reference a and the parameter x would result in a reference to the same location. The description of Algol 60 in the Algol Report, was in this part very operational. The model given was to copy in the argument in all places where the formal parameter was referenced. In this way the body of the above procedure would become -

```
a := a.
```

Some care was necessary in describing the "copy rule" partly because concrete strings were being discussed (cf. the discussion of when parentheses should be inserted). But even using abstract programs, it becomes somewhat tedious to describe this copying (cf. ref. [1]). At least for call-by-reference (see below for call-by-name) there is a simpler, equivalent, mechanism. The idea is to show the sharing by having some auxiliary class of objects and associate both identifiers with the same member of the class. This association is maintained by an environment which maps identifiers into LOCs (chosen in the example below to be suggestive of machine locations). The storage component will no longer associate values directly with identifiers but instead with locations.

What has really been done is to decompose -

```
          DEN
     id --------> VAL
```

into-

```
          ENV          STG
     id --------> LOC --------> VAL
```

but in doing so, the possibility is introduced to have

```
     id1 ----
             |---> n ---> v
     id2 ----
```

so that any change via one of the identifiers is reflected to references via the other. (The use of LOC is, in this model, no more than the expression of an equivalence relation over identifiers).

It is now necessary to consider how a model which has environments handles the block and recursive block problems mentioned above. The locations will be generated so as to be dynamically distinct, so the problem of entering a block and

destroying a denotation which will later be required has certainly been overcome. All that happens is that a new local environment is generated mapping the identifier to a new location (notice such a copying of DEN would be incorrect). In the case of a block which can be known by and called from deeper blocks (i.e. a procedure), it is necessary to show how the base environment, to which it will insert its local bindings, is to be found.

The most economical model would be to assume again that all identifiers are distinct in which case it is possible to show that any valid calling environment contains the required base environment as a sub-part. In this case, then, only the "most recent" existence of any variable can be referred to. This solution does not cover the case where procedures can be passed as parameters! This is precisely because other than "most recent" references are possible. (For a fuller discussion see ref. [7]). The general solution, then, will be to "remember" for any procedure what its base environment should be. In an operational model one would make a procedure denotation contain the pair of procedure text and environment. In a mathematical semantics style definition one would use these two entities to create a function.

The language to be considered is -

E1 proc :: s-nm:id s-parms:id* proc-set s-dcls:id-set st

E2 st = call-st | as-st | ...

E3 call-st :: s-pn:id s-args:id*

E4 as-st :: s-lhs:id s-rhs:expr

Identifiers then correspond either to variables (only one type is considered) or procedures: in the former case a LOC is required, in the latter a procedure denotation, as the associated object.

E5 ENV : id -> (LOC | PROC-DEN)

A not yet initialised value for a variable is allowed, so -

E6 S : LOC -> VAL

E7 LOC = some infinite set

E8 VAL = INTG | ?

A functional type for procedure denotations is given -

E9 PROC-DEN : (LOC | PROC-DEN)* -> (S -> S)

(Notice that goto is not in the current language, so ABN is not required).

In order to cover recursive procedures it is necessary that the denotation of a procedure is available wherever it might (even indirectly) call itself. In an interpretive definition the denotation would have been a pair of the text and the declaring environment. Since denotations here are functional objects, the definition of env' is "recursive". (The validity of such definitions is discussed in ref. [22].)

```
E10   eval-proc-dcl(proc,env) =
        let <id,parm-l,procs,dcls,st> = proc;
        let f(den-l) =
           (let env' = env +
                       ([parm-l[i] → den-l[i] | 1≤i≤lparm-l] ∪
                        [id → eval-dcl(id) | id∈dcls] ∪
                        [s-nm(proc) → eval-proc-dcl(proc,env') |
                                              proc∈procs]);
            int-st(st,env');
            for all id ∈ dcls do
              release(env'(id)))
        result is (f)

      type: proc ENV -> PROC-DEN
```

```
E11   eval-dcl(id):
         let l:alloc();
         assign(l,?);
         result is (l)

      type: id => LOC


E12   int-st(st,env):
         cases st:
         mk-call-st(pn,arg-l) ->
            (let f = env(pn);
             let den-l = <env(arg-l[i]) | 1≤i≤larg-l>;
             f(den-l))
         mk-as-st(lhs,rhs) ->
            (let v:eval-expr(rhs,env);
             assign(env(lhs),v))
         mk...

      type: st ENV =>
```

The functions -

```
E13   alloc : => LOC

E14   release : LOC =>
```

extend and restrict, respectively, the domain of storage. While-

```
E15   assign : LOC VAL =>

E16   contents : LOC => INTG
```

modify and read, respectively, values of storage.

Based on the environment it is possible to clarify two important points. First, it is interesting to note that this is precisely the response of both constructive and mathematical

definitions to the problem of defining a block structure language. This leads to the second point: in what respects is the above manipulation of the environment better than, say "ref. [24]"? The VDL models used the grand state approach and the environment, as well as all "stacked" versions, were contained in the state. This means that, potentially, they can be modified by any function. It was then necessary to show that the interpretation of two successive statements in a statement list was under the same environment. Moreover, such proofs were non-trivial because if the first statement was, for example, a call, the environment was indeed dumped then changed: the proof had to show that by termination of the interpretation of the call, the dumped environment had been restored. The passing of environments as arguments, on the other hand, shows quite explicitly that two successive calls of int-st are passed exactly the same argument. (This was the subject of the, somewhat tedious, proofs of the first two lemmas in ref. [9].)

The language now available is rich enough to discuss the topic of Context Conditions. In the definition given, it is assumed that certain conditions hold for abstract programs which are not expressed by the syntax rules. For example, the definition would simply be undefined for a program which attempted to call a simple variable or which called a defined procedure but with less arguments than parameters. (The attempt to include such type rules in the abstract syntax of ref. [1] was an unnecessary encumberance.) It would, of course, be possible to write appropriate checks into the defining functions. This, however, would not show that the checks, in this language, are of a static nature. That is, it is possible to define a predicate -

is-well-formed : proc -> {true,false}

which only yields true in the case that all such static context conditions are fulfilled. This is not intended to take a position on to what extent type questions in a language should be statically checkable. It is only to argue that it is a useful property of a definition to explicitly show what is static and what can only be checked dynamically. (An

associated point is that it permits freedom to an implementation for programs which contain statically checkable errors in an unexecuted part).

It would be possible to define both blocks and function procedures in the style of this section.

1.6 Further Topics

This section will consider how some other, familiar, language constructs could be tackled in the same spirit as above.

With regard to goto, it is straightforward to extend the last definition to cover goto out of procedure calls. This, along with the merging of the other features already defined, is done in the Appendix. The problem is simple because only known, and therefore most recently activated, instances of labels can be referenced. If the language allows the passing of labels as parameters this property no longer holds. It is now necessary to make each instance of a label dynamically distinct and to do this requires some mechanism like the activation identifiers of ref. [4].

The introduction of label variables (or, in fact, entry variables) into a language brings with it the additional consideration of referencing a label instance which no longer exists. Algol 68 avoids this by constraining the lifetime of any target variable to be not-greater than the lifetime of the label being assigned. PL/I does not make this restriction and so the definition is forced to add a validity check via something like a set of currently active activations.

The definition of call-by-value is simple to include by the creation of a special location which will be the only one affected by any changes via the parameter. This is a close model of the Algol 60 description of assignment to new variables in an imaginary block. The fact that PL/I makes the

choice between call-by-value and call-by-reference on the calling side is shown in ref. [4]. The more powerful call-by-name of Algol 60 is handled via the mechanism for passing procedures.

It is frequently desirable in a language definition to leave the implementer some freedom of order of evaluation. This is a wider freedom than optimisations which guarantee an equivalent result. For instance, it may be reasonable to permit the access of referenced variables in an expression to be made in any order even if the expression contains function references which could cause side-effects. In allowing such orders in a definition the language designer is warning the user not to write programs which rely on any particular order. The question of how to formally define such freedom is an open problem!

First note that in -

(a + (b + c))

the definition may wish to allow not only -

a b c, a c b, b c a, c b a

but also -

c a b etc.

It is not, therefore, adequate to choose one path or the other at each branch: it is necessary to inherit the arbitrariness. The response of VDL to this problem was to make the control component of the machine into a tree. The operations to be performed were put on parallel branches if they were to be executed in any order and the LAMBDA function randomly chose any available leaf of the control for execution.

The definition in ref. [1] was in fact very similar and only achieved its functional nature by building this relevant property of LAMBDA into the definition. Since the only place such arbitrary order could occur was in expression evaluation this was in fact a reasonably small impact.

The definition in ref. [4] shifts the problem to the meta-language by introducing the "," separator. The problem is not solved because the definition of a combinator which provides for the inheritance of sequencing freedom is not provided.

H. Bekič, among others, has pointed out that to combine two pieces of "interfering" program it is necessary to know more than their (extensional) functional meaning. His approach to this problem is discussed in ref. [3]. While this approach may be generally required, the current author would prefer to pursue the definition more axiomatically: in the first place by defining conditions of good cooperation that guarantee non-interference.

One final area, that of Storage Models, brings in the role of axiomatic parts of a definition. In some languages there is great freedom left to the implementor as to what storage mappings of the data structures are required. In PL/I, for efficiency reasons, the programmer can take alternative views of an aggregate and thus the language does somewhat constrain the mappings. Even in ref. [4], however, it was found worthwhile to state the basic storage model abstractly (for example, viewing an array value as a mapping from a (hyper-)rectangle of integers to values, rather than as a linearised list thereof) and to express the additional constraints separately. For a fuller discussion see ref. [2].

1.7 Summary

The difficulty of defining a programming language as distinct from a purely functional language is that changes to a state occur. A functional definition in which functions are read extensionally (cf. section 1.2) is not immediately applicable. Three alternative solutions were discussed; to consider the definition as an interpreting program (ref. [24]); to consider all interpreting functions as having an additional argument and

result which is the state (ref. [1]); to consider the defining functions as producing a functions from states to states (ref. [19]). It is possible, by adopting carefully chosen notation, to write in a style which can be read in more than one way. Except for the problem of arbitrary order, combinators can be provided which permit ref. [4] to be read in all three ways.

The advantages of the different ways of viewing a definition are returned to in the next part of the paper.

Whatever one's chosen view of a semantic definition, there are more important considerations which influence what is written. The central guide-line proposed is that the definition should not possess properties which are not inherent in the language being defined. This is not to say that such definitions are wrong in the overall effect they describe for a program, but rather that a proof is often required that certain details of the model have no effect on the final outcome. If the model can eliminate such details it will facilitate the use envisaged below.

Examples of where definitions can be over-specific range from the use of the grand state approach to trivial items like the use of lists where sets could suffice for components of the state in ref. [25].

Before proceeding to the discussion of proving a compiler correct with respect to a language definition, a few moments should be spent on the question of the correctness of the language definition. As far as "proving" that the definition corresponds to a normal verbal definition, there is little hope. Most such standards descriptions are a mixture of properties (axioms) and partial models for the language. Even if it were possible, which it is not, for the natural language to be read precisely, such definitions would be shown to be at best incomplete, at worst inconsistent.

The direction followed by ref. [25] is, of course, very encouraging in that it is a huge step towards standardising via a document which could be considered to be formal.

In spite of the difficulties in establishing correctness, it is possible to consider some property like consistency. At the most trivial level a definition of the size required by current languages should be checked to be free of clerical errors like passing the wrong number of arguments to a function. Much more subtle are the questions of termination and existence of implicitly defined objects. In ref. [4] an attempt has been made to insert pre, post and assertion comments as an aid to seeing why the authors believe the definition to be "consistent". The question of what a proof would entail is currently under consideration.

2. DEVELOPMENT OF A TRANSLATOR SPECIFICATION

The overall task under consideration is the creation of a program to translate texts of a source language into texts of a machine language. Based on a definition of the source language, this section discusses how to obtain a specification for a mapping from the source to the object language. It is obvious that before this process can begin, an understanding of the object machine is required. The question of whether this understanding must be "formally" documented is returned to below. The step by step development process to be discussed is very similar to that in ref. [15] (and even ref. [9]).

The first question to be resolved is which of the reading styles of the source definition are to be adopted. This question must to some extent remain open until more examples have been fully worked out. There seem, to the current author, to be two arguments for taking the interpretive view of the definition as the basic one. Firstly, it is very unlikely that the case distinctions, which have been shown in the source definition, are exactly those required in the choice of code to be produced. Secondly, a mapping of source programs to functions from states to states has more to be developed than these generated functions: manipulations of, for example, the environment will also require modelling in the object state.

The abstract state of the source definition was chosen to permit a large range of implementations. The time has now come, with a particular machine in view, to be more specific. The designer's task is to find concrete realisations, on his particular object machine, for the abstract state. This development might be a multi-stage process in the case of developing something like an ENV to a display model like those of ref. [7]. Each stage of development provides a new, more concrete, version of an object. This model will have properties not possessed by the more abstract object. For example a list has an ordering property not present in a set. For this reason, the appropriate style to document the relation believed to express the correctness is from (more) concrete to abstract. Thus, if S is a set and modelled by a list L, the set is retrieved by -

retr-S(L) =
{L[i] | 1≤i≤$\underline{1}$L}

type: LIST -> SET

One would then prove that the new function models the old in the same style as discussed in section 3 (this notion is like Milner's "Simulation" in ref. [18]).

Another example of the decisions made during the development of the interpreter, is the removal of arbitrary ordering. The flexibility was permitted by the language to provide freedom for the implementor. Other than those aspects which result in real use of parallel hardware, the randomness is removed in favour of the choice that best fits the compiler design.

The rationale is to attempt always to find a model and express its correctness by showing how, among other irrelevant details, it computes the result of the previous stage. In this way it is possible to avoid large equivalence proofs whose structure is hard to see. In fact, for many stages, documenting the retr functions may itself be an adequate justification. The definition is thus providing, not only the correctness criteria but also a basis for the correctness argument.

At the termination of the work detailed above, which may be multi-stage, there exists an interpreter which functions on a state representable on the object machine. The state transitions, however, are still written in a meta-language. It is now possible to record the machine operations which are believed to produce exactly the same state transformation. In a sense, this can be considered as documenting assumptions about the object machine which may be a more attainable goal than seeking its full formal definition.

It should now be possible to read the interpreter as a mapping by expanding all of the case distinctions which are static in the sense that they depend on the text alone. If the machine operations are inserted, this is now a function from (abstract) source language to the object language.

Such a function will serve as the specification for the development to be described in section 3. It is important to note that the subsequent development requires no understanding of the source language! The meaning of the language was used to justify this mapping. The mapping itself is a specification purely in terms of strings.

It is now appropriate to consider an example. It might have been useful for this section to consider some model of the block concept (cf. ref. [7]), but it will provide a better link to the next section to use the example of compiling expressions.

Consider the language of section 1.3, States are -

1 DEN:id -> INTG

Given, is -

2 apply-op : INTG op INTG -> INTG

Then the definition could be written -

```
3  eval-expr(e):
       cases e:
       mk-inf-expr(e1,op,e2) ->
          (let v1:eval-expr(e1);
           let v2:eval-expr(e2);
           result is (apply-op(v1,op,v2)))
       mk-var-ref(id) -> contents(id)
       mk-const(n) -> n

   type: expr => INTG


4  int-st-l(st-l,i):
       if i ≤ lst-l
          then
             (let mk-as-st(lhs,rhs) = st-l[i];
              let v:eval-expr(rhs);
              assign(lhs,v);
              int-st-l(st-l,i+1))
          else
             I

   type: as-st* INTG =>
```

Now, considering an actual machine, the first problem to note is the way individual values are retained by the "let vi" even though the evaluation of the other sub-expression may itself give rise to many such uses. On most machines this would give rise to some use of temporaries, so consider storage extended so that -

```
5  DEN1 : (id | T) -> INTG
```

where -

```
6  id ≠ T = {}
```

A simple algorithm can now be given which will act as an interpreter on the new class of states. (The author is aware that superfluous assignments are made - but their removal lengthens the example without adding any new concepts).

```
7  trans-expr(e,j):
       cases e:
       mk-inf-expr(e1,op,e2) ->
          (trans-expr(e1,j);
           trans-expr(e2,j+1);
           assign(t[j],apply-op(contents(t[j]),op,
                                contents(t[j+1])))
       mk-var-ref(id) ->
          assign(t[j],contents(id))
       mk-const(n) ->
          assign(t[j],n)

   type: expr INTG =>


8  trans-st-l(st-l,i):
       if i ≤ lst-l
          then
             (let mk-as-st(lhs,rhs) = st-l[i];
              trans-expr(rhs,1);
              assign(lhs,contents(t[1]));
              trans-st-l(st-l,i+1))
          else
             I

   type: as-st* INTG =>
```

To show that the trans-expr interpreter models eval-expr, it is necessary to prove -

```
contents(t[j]) after trans-expr(e,j) = eval-expr(e)
```

If one appends the statement that for i < j, trans-expr(e,j) leaves contents(t[j]) unchanged, it is easy to prove the

combined property by structural induction on the class of expressions. The further extension to as-st lists is trivial.

As suggested above the new interpreter has been shown to model the original defining interpreter by showing how to retrieve the results of the latter from the former.

The type of the function

assign(t[j],apply-op(...))

is DEN1 -> DEN1

Suppose the object machine is of 3 address type (cf. IBM 1401) then, an appropriate instruction might be -

op = ADD - ADD(t[j],t[j],t[j+1])

Inserting such operations, it is now possible to use the function trans-expr as a mapping from source to object languages. It is important to remember that this has been derived step by step from the definition.

(At the risk of labouring the point, it could be remarked that had the statement counter been made part of a grand state, it would now pose problems because there is no model for it in the object state).

It should be clear from the methods used so far that not only in the source definition, but also in the subsequent development it is likely to lead to more work if unnecessary properties are introduced. This, however, touches on one of the problems which has not really been solved. In a large problem the mapping is likely to be such that it cannot be expressed conveniently in a "single pass". If a multi-pass mapping is described and its structure differs from that of the eventual translator, the proof is likely to be much more difficult. More work is needed in this area.

Before concluding this section it is worth considering what happens if a defining model is chosen, or given, which is in some areas of the language too concrete. That is, there are some details present which do not appear in the planned model. Not only should one refrain from throwing the definition away; one should also not embark on a complete equivalence proof. It has been shown in ref. [10] how one can, as a development stage, introduce a more abstract notion than that of the source definition for one area of the language. One can then prove that satisfying the new abstract notion ensures overall equivalence. The subsequent development can now be made from the more abstract definition. In this respect it would be interesting to prove that the storage model of ref. [25] was a model of that in ref. [4].

3. FORMAL DEVELOPMENT

Faced with a program specification and a code listing it is difficult to ascertain whether the latter satisfies the former. The basic intention of Formal Development is to provide a framework in which the design can be recorded step by step. Thus, the idea of top-down documentation of a development is subscribed to. In addition it is argued that each level of development can be documented precisely enough that its correctness can be the subject of a proof.

It is important to distinguish the current proposal from the idea of writing a program then constructing a proof that it fulfills its specification. The possibility to use abstraction during a development makes the construction of a step-wise proof far more tractable. Furthermore, the amount of work to be redone, when an attempt to construct a proof uncovers an error, is reduced.

It may also avoid misunderstanding if it is stated right away that Formal Development is not proposed as a rule to order invention. The backtracking and effect of inspiration conveyed

by ref. [20] are much more typical of program invention. But, just as one frequently rearranges a proof when writing it up, it is worth documenting a design so that a reader can see a coherent development of ideas.

Two main sorts of development steps are discussed. The first is the one normally connected with top-down development. Given a specification of what is required, one writes some sequence of operations which show how the task may be achieved. The operations may be either statements of some language or assumed sub-operations: in either case their specifications are also recorded. Since some formal notation is being used it is now possible to write down why one believes the combined sub-operations perform the given task.

The second sort of development step comes from the wish to use abstractions of normal data objects. By using objects which only have the properties relevant to an algorithm, it is possible to drastically reduce the length of both specifications and correctness arguments. At some point in the development it becomes necessary to seek an efficient representation which can be described in the language being used: such steps of development are considered in section 3.2.

3.1 Operational Abstraction

Operations are considered to be transformations on states from some class, say Σ. For some operation, say OP -

$$\sigma \; [OP] \; \sigma'$$

means that OP will produce a state σ' when "run in" a state σ (only deterministic operations are considered in the current paper).

The term abstraction is applied because operations, which may not be available (other than by a yet to be performed

construction) can be discussed. They are in fact discussed via an implicit specification. The definition of an operation will be given via two predicates one which specifies the domain over which it must yield a result -

$$\text{pre-OP} : \Sigma \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$$
$$\text{pre-OP}(\sigma) \supset (\exists \sigma')(\sigma\ [\text{OP}]\ \sigma')$$

and the other of which specifies the input/output relation required for the operation -

$$\text{post-OP} : \Sigma\ \Sigma \rightarrow \{\underline{\text{true}}, \underline{\text{false}}\}$$
$$\text{pre-OP}(\sigma) \wedge \sigma\ [\text{OP}]\ \sigma' \supset \text{post-OP}(\sigma, \sigma')$$

If both of these conditions hold the fact is recorded by -

$$\text{pre-OP}\ \langle \text{OP} \rangle\ \text{post-OP}$$

These two predicates, then, record everything necessary about the meaning of OP (there is of course nothing about performance etc.).

Suppose a specification is given in this form, how does one proceed? The problem is decomposed to sub-problems by choosing a set of (simpler) operations, which, if one had them, could be combined in some stated way to fulfil the specification. The assumptions on the sub-operations will be recorded in exactly the same style. Eventually all of the sub-operation assumptions will be true of statements in the language being used. Until that time they provide specifications for further work.

The most trivial way of combining two operations in most languages is to separate them with ";" showing that the execution of the first is to be immediately followed by execution of the second. The conditions necessary to show that such a combination of the two operations -

pre-OP1 <OP1> post-OP1
pre-OP2 <OP2> post-OP2

will satisfy -

pre <OP1;OP2> post

are firstly that each operation will only be used over its stated domain -

pre(σ1) ⊃ pre-OP1(σ1)
pre(σ1) ∧ post-OP1(σ1,σ2) ⊃ pre-OP2(σ2)

and secondly that the overall input/output relation can be derived from the combination -

pre(σ1) ∧ post-OP1(σ1,σ2) ∧ post-OP2(σ2,σ3) ⊃ post(σ1,σ3)

(The adequacy of these conditions is proved in ref. [12]). For such a simple combination, the requirement to prove three lemmas appears excessive. In many situations however, special cases can be applied. For instance, with total operations (pre = true) the first two lemmas are vacuously true. Furthermore, it is certainly not being suggested that every use of ";" in a program should be accompanied by formal proofs: but a check list has been provided to which an appeal can be made in case of doubt.

The reader should observe that a specification is passed on to the next stage of development which states all of the properties relied on. Thus it is not necessary to later show that the development of that operation does not disturb the current proof. There is a complete split of the problem of providing justifications.

More methods of combining operations are defined in the same style by ref. [12], section 9 of that paper also considers how the set could be further extended.

3.2 Representational Abstraction

In many respects this might be the more important of the two forms of abstraction being considered: it is certainly the one which is under-employed.

The idea of data abstraction has, in fact, already been used in the earlier parts of the paper. The language definitions and mappings have both used an abstraction of the program text (i.e. that class of objects described by the abstract syntax).

That this was necessary can be seen by considering the alternative of redefining these functions in terms of concrete strings.

If then, it is difficult to even state the specification in terms of detailed data representations, it will become impossible to write arguments for correctness at such a level of detail.

The proposal is that development of an algorithm should only bring in those properties of the data structure that have an effect on the algorithm. That this permits a reasonable percentage of the development can be seen in the example of section 3.4 below. Thus one is able to postpone fixing the representation of a data object until adequate reason (e.g. the performance of a sub-algorithm) can be ascertained.

The interface between operations might, then, be described in terms of sets or maps for example: in the final code linked lists or hash tables might be the chosen representations.

It is necessary to discuss what goes on in a development step which refines a data representation. Essentially what one is doing is adding properties to the data structure (e.g. the list has an ordering property not present in the set). It is possible to <u>retrieve</u> all of the data of the abstract level from the more concrete.

Suppose an operation on states of class D has been used, such that -

pred <OPd> postd

and one now wishes to show that -

pree <OPe> poste

is an adequate simulation. It is sufficient to find a relationship between the two state classes -

retrd : E -> D

which shows that OPe works on a wide enough class of states -

pred(d) ∧ retrd(e)=d ⊃ pree(e)

and that the new operation produces states matching (under retrd) those produced by the old operation -

pred(d) ∧ retrd(e)=d ∧ poste(e,e') ⊃ postd(d,retrd(e'))

(This notion differs from that in ref. [18] in that the Operations are not, necessarily, functions).

3.3 Example of Expression Compilation

The input/output relation given for trans-expr in section 2 is defined to operate on objects of type "expr". These were conveniently chosen to be tree representations of the original linear form (presumably infix). Without going all of the way to considering the parsing and tokenising of an external string, consider a reverse-polish text which might result from such a first parse -

```
c-expr ::= c-inf-expr | c-var-ref | c-const

c-inf-expr ::= c-expr c-expr c-op

c-var-ref ::= ...

c-const ::= ...
```

The relation of this to the class expr can be specified by a retrieve function which uses a stack -

```
retr-expr(tl) =
   for i = 1 to l tl do
      (is-c-var-ref(tl[i]) ->
         push(retr-var(tl[i]))
       is-c-const(tl[i]) ->
         push(retr-const(tl[i]))
       is-c-op(tl[i]) ->
         (let e2:pop;
          let e1:pop;
          push(mk-expr(e1,retr-op(tl[i]),e2))));
   result is (pop)

type: c-expr -> expr
```

Not only does this retrieve function give the correctness criterion for the following translate function, the stacking is suggestive of a way to track the temporaries.

Assuming an external variable b -

```
trans-c-expr(tl):
   for i = 1 to l tl do
      (is-c-var-ref(tl[i]) ->
         (b := b + 1;
          assign(t[b],contents(retr-var(tl[i]))))
       is-c-const(tl[i]) ->
         (b := b + 1;
          assign(t[b],retr-const(tl[i])))
       is-c-op(tl[i]) ->
         (assign(t[b-1],apply-op(contents(t[b-1),
                                 retr-op(tl[i]),
                                 contents(t[b])));
          b := b-1)

type: c-expr =>
```

The correctness can now be proved if -

```
b := 0; trans-c-expr(tl)
            ~ trans-expr(retr-expr(tl),1)
```

This result follows from a proof, by induction on the length (and possible constructions of) tl, of the stronger statement -

```
b := k; trans-c-expr(tl)
        leaves b = k + 1 and creates the same as
            trans-expr(retr-expr(tl),k+1)
```

3.4 The Earley Example

The rather short treatment of Formal Development offered may leave the reader unclear as to how a bigger example looks. Whilst the notation used in ref. [12] is thought to have correctly made the step from development via functions to development via operations, the example of that report is

unconvincing. This is mainly because the algorithm considered was so oriented to arrays that the use of an abstract data representation is somewhat artificial.

The example of ref. [11] is more interesting with respect to data abstraction and a short outline of a rewrite of its development is now given -

Specification: find a (general, table driven recogniser) algorithm -

REC : grammar nt symb* -> {YES,NO}

Where the abstract form of a grammar is -

```
grammar : nt -> rhs-set
rhs = el*
el = symb | nt
```

The pre-condition defines that there are rules for each non-terminal and that there is exactly one rule for the sentence non-terminal. The post-condition states that REC should yield "YES" if and only if the symbol string can be produced from the given grammar. ("Produceable" is defined).

Step 1: Splits the problem into an input stage which stores the grammar; a main stage which creates "State Sets" which will contain information on all possible top-down parses; an output stage which yields YES or NO depending on a predicate of the state sets. The storage for the grammar is still specified in terms of the (abstract) map. This may be a disappointment to the programmer assigned the task of constructing the input routine since he has little to work on yet. But the cost of fixing this interface for his convenience is that the far more time consuming parsing operation has not yet been developed far enough to get his views on an efficient representation.

The state sets are also described abstractly (as a list of sets of tuples) since the purpose of this stage is to show that certain upper and lower bounds on the state sets are sufficient

to make the final predicate correct. Notice this extreme form of abstract definition. There is a great deal of freedom in the given bounds and different algorithms could be constructed to use this freedom. (In fact the specification has been of considerable use in considering optimisations).

Step 2: Introduces Earley's operations (Prediction, Completion, Scanning) which generate new states. The state sets are defined as the minimum sets satisfying a certain equation. Such sets are shown to fall within the bounds stated in Step 1. Notice that not only are these operations defined in terms of abstract data objects, they are also implicitly defined. This is in distinction to ref. [8] in which the algorithms are programmed with operations on the abstract data: this form of development is employed later.

Step 3: Begins to consider representations by mapping state sets onto state lists. But notice that, since a list of lists is not a convenient data object in a von Neumann machine, the step to a concrete representation is not yet complete. Using the chosen algorithm on lists it is necessary to introduce a restriction (no zero length rhs) to the allowable grammars. It would, however, have been possible to use a different algorithm at <u>this</u> stage of development and avoid the restriction.

Step 4: Makes a similar ordering step to the data structure representing grammars.

At this point most of the algorithm, as such, is designed and it is clear what the common operations on the data structures are. Now is the time to give the concrete (PL/I) data objects! (In fact those used were quite complicated BASED variables with the REFER option.) Doing this prompts a macro style of data abstraction like ref. [5] which is similar to that used in ref. [8].

4. SUMMARY

The aim of the paper has been to show how a large problem, in this case the development of a compiler, can be decomposed into small steps. Providing each step is adequately documented, a complete design history is thus obtained. One of the views expressed is that each stage of development should be supported by a justification. This implies that steps of design are recorded in a notation on which it is possible to base a correctness argument. Such correctness arguments are sought with a view to human readers rather than mechanical theorem checkers.

The key to making such an approach practical is the use of abstraction. In each of the sections the value of stating the minimum properties has been shown. Although it is frequently more difficult to find an appropriate abstraction than to provide a construction, the advantages of the former make the effort worthwhile.

By employing both data and operational abstraction, a view of the development process is obtained where the successive stages are realisations of the same algorithm at ever greater levels of detail. Taking this view, the normal style of correctness argument is based on showing how the (more) abstract model can be retrieved from the (more) concrete realisation. In this way it is possible to avoid general equivalence proofs.

The requirements of a design language to be able to specify operations implicitly and to use very abstract data objects are very different from those of programming languages. Although a particular notation (that of ref. [4]) has been employed as the design language, it should be emphasised that it is the method not a particular notation which is being proposed here.

Acknowledgement

Most of the ideas contained in this paper were developed in collaboration with members of the Hursley and Vienna IBM Laboratories. The members and meetings of IFIP WG 2.3 have also been a great stimulus.

References

[1] C.D.Allen, D.N.Chapman and C.B.Jones, "A Formal Definition of Algol 60", IBM Hursley Technical Report, TR 12.105 August 1972.

[2] H.Bekič and K.Walk, "Formalisation of Storage Properties" in "Symposium on Semantics of Algorithmic Languages" (Ed.) E.Engeler, Springer-Verlag Lecture Notes in Mathematics No. 188, October 1970.

[3] H.Bekič, Presentation on "Semantics of Actions" given at Newcastle University, September 1974.

[4] H.Bekič et al "A Formal Definition of PL/I" to be printed as a Technical Report of IBM Laboratory Vienna.

[5] A.Hansal, "Software Devices for Processing Graphs Using PL/I Compile-time Facilities", Info Proc Letters. 1974.

[6] W.Henhapl and C.B.Jones, "On the Interpretation of Goto Statements in the VDL", IBM Vienna Note, LN 25.3.065, March 1970.

[7] W.Henhapl and C.B.Jones, "The Block Concept and Some Possible Implementations, with Proofs of Equivalence", IBM Vienna Technical Report, TR 25.104, April 1970.

[8] C.A.R.Hoare, "Proof of Correctness of Data Representations", Acta Informatica, Vol. 1, pp 271-281, 1972.

[9] C.B.Jones and P.Lucas, "Proving Correctness of Implementation Techniques", in "Symposium on Semantics of Algorithmic Languages" (Ed.) E.Engeler, Springer-Verlag Lecture Notes in Mathematics No. 188, October 1970.

[10] C.B.Jones, "Sufficient Properties for Implementation Correctness: Assignment Language", IBM Hursley Note, TN 9002, June 1971.

[11] C.B.Jones, "Formal Develcpment of Correct Algorithms: An Example Based on Earley's Recogniser", presented at ACM SIGPLAN Conference, SIGPLAN Notices Vol. 7, No.1, January 1972.

[12] C.B.Jones, "Formal Development of Programs", IBM Hursley Technical Report, TR 12.117, June 1973.

[13] P.J.Landin, "A Correspondence Between Algol 60 and Church's Lambda-Notation: Part I", Comm. of ACM, Vol.8, No.2, February 1965.

[14] P.Lucas, "Two Constructive Realisations of the Block Concept and their Equivalence", IBM Vienna Technical Report, TR 25.085, 1968.

[15] P.Lucas, "On Program Correctness and the Stepwise Development of Implementations", presented at IBM Conference at Pisa University, 1972.

[16] P.Lucas and K.Walk, "On the Formal Description of PL/I" in Annual Review in Automatic Programming, Vol.6, Part 3, Pergamon Press, 1969.

[17] J.McCarthy, "Towards a Mathematical Science of Computation" presented at IFIP Congress 1962.

[18] R.Milner, "An Algebraic Definition of Simulation Between Programs", Stanford University AIM-142, February 1971.

[19] P.Mosses, "The Mathematical Semantics of Algol 60", Oxford University Computing Laboratory, PRG-12, January 1974.

[20] P. Naur, "An Experiment on Program Development", BIT 12, pp 347-365, 1972.

[21] J.C.Reynolds, "Definitional Interpreters for Higher-Order Programming Languages", presented at 25th National ACM Conference, August 1972.

[22] D.Scott and C.Strachey, "Toward a Mathematical Semantics for Computer Languages", in "Proceedings of the Symposium on Computers and Automata", Microwave Research Institute Symposia Series Vol.21, Polytechnic Institute of Brooklyn, 1971.

[23] C.Strachey, "Continuations: A Mathematical Semantics which can deal with Full Jumps", unpublished.

[24] K.Walk et al, "Abstract Syntax and Interpretation of PL/I", IBM Vienna Technical Report, TR 25.098, 1969.

[25] "PL/I BASIS/1" ECMA ANSI working document, February 1974.

Appendix

This appendix contains a definition of the language obtained by merging the separate features of section 1. The definition is written in a style (cf. the type clauses) which can be read as mathematical semantics.

ABSTRACT SYNTAX

```
proc     :: s-nm:id s-parms:id* proc-set s-dcls:id-set cpd-st
st       =  as-st | goto-st | call-st | cpd-st
as-st    :: s-lhs:id s-rhs:expr
goto-st  :: id
call-st  :: s-pn:id s-args:id*     /*args dcl, proc or parm*/
cpd-st   :: nmd-st*
nmd-st   :: s-nm:[id] s-body:st
expr     =  inf-expr | var-ref | const
inf-expr :: expr op expr
var-ref  :: id
const    :: INTG

id   )       not further specified
op   )
INTG )
```

DOMAINS

ENV: id -> (LOC | PROC-DEN)

S: LOC -> VAL

LOC = infinite set

VAL = INTG | ?

PROC-DEN: (LOC | PROC-DEN)* -> (S -> S ABN)

ABN = [id]

FUNCTIONS

```
eval-proc-dcl(proc)(env) =

   let <id,parm-l,procs,dcls,mk-cpd-st(ns-l)> = proc;

   let f(den-l) =
     (let env' : env +
                 ([parm-l[i]  → den-l[i] | 1≤i≤lparm-l] u
                  [id         → eval-dcl(id)  | id∈dcls] u
                  [s-nm(proc) → eval-proc-dcl(proc)(env')  |
                                                  proc∈procs]);
      (trap exit(lab) with
        (free(dcls,env');
         exit(lab));
       int-ns-l(ns-l,1)(env'));
      free(dcls)(env'));

   result is(f)


type: proc -> (ENV -> PROC-DEN)




eval-dcl(id):

   let l: alloc();
   assign(l)(?);
   result is(l)


type: id -> (S -> S LOC)
```

```
free(dcls)(env):

   for all id∈dcls do
      release(env(id))


type: id-set -> (ENV -> (S -> S))




int-st(st)(env):

   cases st:

   mk-as-st(lhs,rhs) ->
     (let v: eval-expr(rhs)(env);
      assign(env(lhs))(v))

   mk-goto-st(lab) ->
      exit(lab)

   mk-call-st(pn,arg-l) ->
     (let f = env(pn);
      let den-l = <env(arg-l[i]) | 1≤i≤larg-l>;
      f(den-l))

   mk-cpd-st(ns-l) ->
      int-ns-l(ns-l,1)(env)


type: st -> (ENV -> (S -> S ABN))
```

```
int-ns-l(ns-l,i)(env):

   if i≤lns-l
      then
        ((trap exit(lab) with
            if is-contained(lab,ns-l)
               then cue-int-ns-l(ns-l,lab)(env)
               else exit(lab);
          int-st(s-body(ns-l[i])))(env));

         int-ns-l(ns-l,i+1)(env))
      else
         I


type: nmd-st* INTG -> (ENV -> (S -> S ABN))




cue-int-ns-l(ns-l,lab)(env):

   let i = (∪i)(is-contained(lab,<ns-l[i]>));
   if lab = s-nm(ns-l[i])
      then int-ns-l(ns-l,i)(env)
      else ((trap exit(lab) with
                if is-contained(lab,ns-l)
                   then cue-int-ns-l(ns-l,lab)(env)
                   else exit(lab);
             cue-int-ns-l(s-body(ns-l[i]),lab)(env));
            int-ns-l(ns-l,i+1)(env))


type: nmd-st* id -> (ENV -> (S -> S ABN))
```

```
eval-expr(e)(env):

   cases e:
   mk-inf-expr(e1,op,e2) ->
     (let v1: eval-expr(e1)(env);
      let v2: eval-expr(e2)(env);
      result is(apply-op(v1,op,v2)))
   mk-var-ref(id) -> contents(env(id))
   mk-const(n) -> n


type: expr -> (ENV -> (S -> S INTG))




is-contained: id nmd-st* -> B

apply-op: INTG op INTG -> INTG


alloc: -> (S -> S LOC)

release: LOC -> (S -> S)

assign: LOC -> (VAL -> (S -> S))

contents: LOC ->(S -> S INTG)            /* ? yields error */
```