The University of Warwick

THEORY OF COMPUTATION

REPORT NO. 23

PROGRAMMING CONSTRUCTS FOR NONPROCEDURAL LANGUAGES

BY

WILLIAM W. WADGE

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND.

March 1978

PROGRAMMING CONSTRUCTS
for
NONPROCEDURAL LANGUAGES

William W. Wadge

Abstract: In this paper how a pure denotative (nonprocedural) language based on the lambda calculus can be provided with purely denotative analogs of the various constructs - such as while loops, blocks, case statements and the like - of conventional imperative structured programming languages. They can be simulated quite adequately using only two simple tools: phrases, which are compound expressions not unlike blocks, and pronouns, special variables not unlike keywords between which certain relationships are 'understood' to hold.

Computer Science Department, University of Warwick, Coventry UK

This paper was presented at the 3rd International Symposium on Programming, Paris, March 1978.

0. Introduction

A denotative (or nonprocedural) programming language is a language based on some mathematical formal system, such as the λ-calculus or predicate logic. A program in such a language is a term or formula in the formal system, rather than a sequence of commands to be executed, and its meaning is the value of the expression or the solution(s) of the formula, rather than the result of obeying the commands. The great advantage of denotative languages is that the transformation or inference rules of the formal system can be applied directly to programs, and are powerful rules for program verification or massage.

The merits of the denotative approach have been well understood for at least a dozen years now (see the discussion following Landin's "The next 700 programming languages", Landin [3]), but at present few computer scientists would consider them to be serious alternatives to conventional imperative languages. This is in part due to some simple misunderstandings (for example, the widespread belief that computation is inherently a sequential activity); but there are also more serious objections to denotative languages, in matters both of form (syntax) and meaning (semantics).

The main objection on semantic grounds is that programmers using a denotative language have no control over the behaviour (as opposed to the meaning) of their programs, ie they cannot make anything happen the way they want. The conclusion: that denotative programming is inherently inefficient.

And the main objection on syntactic grounds (the one with which this paper is concerned) is that denotative languages are too impoverished of expression, lacking as they do the rich variety of constructs - blocks, for loops, case statements and so on - found in imperative languages. The conclusion: that denotative languages are inherently clumsy. It is apparent that this objection is closely related to the former - the various constructs of imperative languages are operationally motivated (so it seems), ie designed to bring about desired activity. Therefore (so it seems) they cannot have denotative analogs.

Now it is certainly true that the definition of the semantics of denotative languages are purely mathematical in that they describe only the denotations of programs and parts of programs. But there is no reason why such a languge (or subset thereof) cannot have an alternate, operational semantics which describes the meaning of programs in terms of activity they give rise to. As long as the operational semantics is equivalent to the mathematical one the programmer may use either (or both) as a programming guide. Moreover, if the implementation is based on the operational semantics, the programmer can use his knowledge of it to bring about desired efficient behaviour by varying the form of his program.

Consider, for example, the case of blocks and procedure (ie. function) declarations. Most Algol programmers have a very operational understanding of these constructs, one in which blocks are entered and exited, storage is allocated and deallocated, and threaded stacks bob up and down. Now Landin in the paper cited described a pure nonprocedural language ISWIM which allows both block structures and function definitions in a very natural (and very simple) way. Yet neither the syntactic rules of the language, nor the formal semantics, nor the rules of inference, make any reference to stacks, pointers, return links and so on. Indeed no notion of 'storage' is required. Of course, there is nothing to stop a programmer from imagining an ISWIM block being 'entered', or a function being called, but it is not necessary to think operationally like that. Even the implementor might want to avoid the classical Algol view of execution, perhaps by using the copy rule for functions and by translating blocks into parts of a data flow network.

Similarly, there is no reason why imperative languages should have a monopoly of conditional, case, and switch statements and the like. It is very easy to devise a simple denotative case expression, and the usual operational view involving "testing" and "jumping" might or might not be helpful to the user. Even repetitive constructs (for loops, while statements and so on) fit in to this framework if we use Lucid's denotative approach to iteration.

In this paper we give details of the denotative constructs just mentioned. What is especially unusual about our approach is that it does not require adding any new syntactic constructions. Instead, the effect can be achieved using only special 'keyword' variables called pronouns among which certain implicit relationships (in the form of equations) are assumed to hold.

1. Phrases

In this paper the formal system on which we base our programming language is the λ-calculus. Now it is certainly true that the pure λ-calculus by itself must, on syntactic grounds alone, be rejected as a practical programming language. All but the simplest λ-expressions are (when presented in full) almost unreadable (even if infix notation and other minor sugarings are allowed). This unreadability is often blamed on the tendency of λ-expressions to be deeply nested; but a more serious problem is the frequently large textual distance between variables and the expressions to which they are bound. For example, even in the expression

$$\{\lambda x\ \lambda y\ x^2+y^2\}(a-b)(a+b)$$

the connection between the variable "y" and the expression "a+b" does not leap to the eye.

Of course, if we are interested in the λ-calculus only as a formal system, ie.

if we are interested only in reasoning about λ-expressions, then these observations are of no consequence. But if we are interested in adapting the notation in a programming language, ie if we are interested in actually reading and writing λ-expressions, then problems of form like the one just discussed are very important.

Fortunately, there is a very simple solution to the problem. It involves adding a class of special expressions (which we call phrases) which include equations defining variables; a variable and the expression to which it is bound are brought together across an "=" symbol. Phrases were first formalized by Landin (in [3]), although (as he points out) mathematicians have been using them informally for a long time. Here are two examples of the above λ-expression in phrase form:

```
let x = a+b in
 let y = a-b in
          x^2+y^2
```

and

```
x^2+y^2 where x = a+b
            and y = a-b.
```

The semantics of phrases can be given by a straightforward translation. For example, if v is a variable and M and N are expressions, then

```
let v = N in M
```

is defined to be

```
{λvM}N.
```

Of course the fact that phrases can be translated into λ-expressions does not in itself justify the claim that the enlarged language is denotative; because in general translations are defined in terms of the form of expressions. What is more persuasive is the fact that there exist rules of inference for transforming phrases so that it is possible to reason about such expressions without reference to their translations. Furthermore, it is also possible to define the denotation of such expressions in a given environment in a straightforward way, also without considering translations.

So powerful are these simple constructs, that if we add them to an applied version of the λ-calculus (ie one which has additional symbols such as "+" together with an interpretation of these symbols); and if we allow infix operators and the like; we have a perfectly usable (and very expressive) programming language.

2. The pronoun "result"

All the phrases mentioned so far have, syntactically speaking, two parts: a list of definitions, and an expression to which the definitions are to be applied. As our first example of the use of pronouns we define a syntactically simpler phrase which consists solely of a list of definitions.

The phrase we are describing might be called the "applicative valof". A phrase of this type consists of a set of equations enclosed by the 'brackets' "valof" and "end". Each equation is of the form

variable = expression

and no variable may occur as the left hand side of two different equations in the same valof. The variables appearing on the left hand side of equations in a valof are called the locals of the valof; and among these must be the special variable "result".

The following is a typical valof phrase

```
valof
     X = car A
     Y = cdr B
     C = cons(Y,B)
result = cons(X,C)
end.
```

The following are not legal valof phrases.

```
valof
     X = A - B
     Y = X + 1
     Y = Y + 1
result = X
end
```

```
valof
     Q = A + B
     R = A - B
     Z = Q · R
end
```

the first because the variable "Y" has two definitions, and the second because the special variable "result" is not defined.

The meaning of a valof phrase is, informally at least, clear enough: it is the value of "result" in the environment defined by the body (recursive definitions are allowed). It is possible to make this definition precise by giving a rule for translating valof expressions into pure λ-expressions, but we have already remarked that the existence of translations does not necessarily justify the use of the adjective "applicative". Instead, we give a direct semantics and mention some rules for transforming valof's.

Let us assume that we already have in mind some structure, ie some domain of possible values together with interpretations of constant symbols like "+" or "cons" over the domain. Then given any environment σ (ie any function which maps variables into elements of the domain) we define the meaning of the phrase

```
valof
  v1 = E1
  v2 = E2
  ...
  vn = En
end
```

to be σ'("result") where σ' is the least defined environment such that

(i) σ' agrees with σ except possibly on the values assigned to the locals $v_1, v_2, \ldots v_n$;

(ii) σ' makes valid all the equations in the body of the valof, ie $\sigma'(v_i)$ is, for each i, the value of E_i in σ'.

Two of the most important rules which can be derived from the semantics just given are the substitution rule and the renaming rule. The first says that if a phrase contains the equation $v = E$ then E may be substituted for any occurrence of v which is free for E in any expression which is the right hand side of an equation in the phrase. This rule is related to the β-rule of the λ-calculus, and it justifies the use of the symbol "=". The second rule, the renaming rule, allows us to replace all free occurrences of a local v by a variable v' provided (i) v' does not occur free in any equation in the phrase; (ii) all free occurrences of v are free for v'; and (iii) v is not the variable "result". This last restriction extends to all the pronouns to be introduced, and brings out clearly the difference between them and ordinary local variables. Pronouns may not be renamed because they have particular, predefined roles to play - in the case of "result", to define the value or output of the phrase.

3. Case statements

The introduction of phrases does not remove the only source of unwieldly and unreadable expressions. Some of the worst examples occur (and not just in denotative languages) when if-then-else expressions are deeply nested. A good example appears in the following recursive definition of a function comp which tests S-expressions for equality:

```
comp = λx λy  if atom(x) then if atom(y) then eq(x,y)
                else false else if atom(y) then false
                else if comp(car x, car y) then comp(cdr x, cdr y)
                else false.
```

Of course writing such an expression is simply not good style. A far better way is to rewrite the big expression as a phrase with locals defined as tests and corresponding results. Pronouns allow us to do this in a systematic way and at the same time simulate a kind of logical case statement.

Therefore we add to our list of unrenameable special variables those in the three sequences

test0, test1, test2, ...
res0 , res1 , res2 , ...
cond1, cond2, ...

We expand the class of valid phrases by allowing variables from the first two (but not the third) list to be defined in phrase bodies; and by allowing, for each i, the i^{th} cond variable to be used provided the phrase has definitions of every test variable up to the $i\text{-}1^{th}$ and every res variable up to the i^{th}. The following relationship between these variables is understood to hold: the value of condi is, for each i, the value of resj for the least j less than i for which testj is true; and if all are false, condi is resi. Our definition of comp can now be rewritten:

```
comp = λx λy valof
                result = cond3
                 test0 = atom(x) ∧ atom(y)
                 res0  = eq(x,y)
                 test1 = atom(x) ∨ atom(y)
                 res1  = false
                 test2 = comp(car x, car y)
                 res2  = comp(cdr x, cdr y)
                 res3  = false
               end
```

When we say that the above relations are "understood" to hold we can make this precise in two ways: syntactically, as meaning that the equations defining condi (such as

```
cond3 = if test0 then res0 else if test1 then res1
        else if test2 then res2 else res3 )
```

are considered as automatically included in the body of every phrase; or semantically, as meaning that we consider only those environments which make such equations true. Note that there are two types of pronouns: independent pronouns like "result" which can be defined by the programmer, and dependent pronouns like "cond3" which cannot be defined but may be used provided there exist definitions for those other pronouns upon which the dependent pronoun is understood to depend.

The pronouns and relationships just described give us the effect of a kind of logical case statement. It is easy to devise other pronouns and relationships which give us the effect of an integer case statement. We add as independent pronouns those in the list

case0, case1, case2, ...

plus the variable "default". And we add as dependent pronouns those in the list

switchon0, switchon1, switchon2, ...

representing functions of one integer argument. The understood relationship between these pronouns is that

```
switchoni =  n if n eq 0 then case0 else if n eq 1 then case1
             ... else default.
```

The pronoun switchoni may therefore be used provided the phrase has a definition of casej for every j less than i.

Here is a typical phrase using these pronouns:

```
valof
  result = switchon3(opcode)
   case0 = A + B
   case1 = A · B
   case2 = - A
 default = err(8)
end
```

Sequences of pronouns (like the "case" sequence) could be replaced by single function pronouns of a natural number argument although this means that some evaluation must be performed just to check whether or not a phrase is well formed. One way to make sure that this is always possible is to require that the arguments be constants (eg "case(2)"); but it is also sufficient to require that the arguments depend only on variables defined equal to constants in some enclosing phrase. This last approach would allow

```
valof
  plus = 0
 times = 1
 minus = 2
       ...
    valof
         result = switchon3(opcode)
      case(plus) = A + B
     case(times) = A - B
     case(minus) = - A
         default = err(8)
    end
       ...
end
```

and gives us the effect of the "manifest" constants of imperative languages. Clearly other extensions, such as simultaneous switches on two or more expressions, are easily formulated.

4. Loops

Many of the constructs of imperative languages are (like the classic Algol for statement) loop-related constructs: they are used by programmers to bring about and control repetition. Now strictly speaking a programmer using a denotative

language cannot 'bring about' repetition or any other kind of operationally defined activity; but by using the approach of the language Lucid (see Ashcroft and Wadge [1]) it is possible to write denotative programs which may be understood in terms of iteration. Pronouns are a valuable addition to the Lucid approach because, as we will see, they allow us to simulate the imperative loop constructs.

Lucid achieves its effect by having variables and expressions denote time sequences of data so that time dependent functions such as first and next can be defined denotatively. The unary functions first and next (or the binary function fby ("followed by")) are used to give inductive definitions, and the binary function asa ("as soon as ") extracts values from time sequences. Lucid itself is defined in terms of clauses (compound assertions), but we will use an obvious analogous phrase oriented version. Here is a simple program to compute the integer square root of the constant N:

```
valof
   I = 0 fby I + 1
   J = 0 fby J + 2 · I + 1
result = (I-1) asa J>N
end
```

Lucid iterations never terminate; they are (conceptually) infinite computations from which values are extracted using asa. Explicit use of asa is not always convenient (especially in proofs), and so we show how pronouns can be used to simulate the more conventional while construct. We use an independent pronoun "halt" and a dependent function pronoun "last" of which it is understood

```
last = λx  x  asa halt.
```

The above loop can be rewritten as

```
valof
   I = 0 fby I + 1
   J = 0 fby J + 2 · I + 1
  halt = J>N
result = last(I) - 1
end
```

These pronouns are especially useful when it is necessary to refer to the last value of more than one expression,eg

```
if last(X)<M then last(Y) else last(Y)+1
```

would be somewhat less clear if written in terms of asa.

We can also simulate the classic for-loop with pronouns. We add an independent pronoun "range" and two dependent pronouns, "index" and "final" of which it is understood:

```
index = car(range) fby index + 1
final = λx  x asa index eq cdr(range)
```

(we are assuming a step size of one). Here is a simulated for-loop which computes the vector dot product of a(1), a(2), ... , a(N) and b(1), b(2) , ... , b(N): (we are using "," to denote a pair constructor)

```
valof
     range = (1,N)
         I = index
       sum = a(1)·b(1) fby sum + next a(I)·b(I)
    result = final(sum)
end
```

A very important property of loops like the one above is that they are guaranteed to terminate (provided N is defined).

A frequent criticism of the classical for- and while- constructs is that in each case the loop can terminate in only one way; whereas in 'real life' there are iterative algorithms which seem to allow several different ways of terminating, each giving different results. A classic example is searching a table for an item which may not be present (two exits), and another is comparing two sequences to determine their relationship in the lexicographic ordering (three exits). Whether coded as for- or while- loops or even in Lucid with last, the resulting programs are very inelegant because of the need to find out, after the loop has terminated, exactly how it was terminated. Some authors (eg Knuth in [3]) have used these examples to argue that the goto statement might have a place in structured programming. Fortunately pronouns provide a simple (and, needless to say, goto-less) solution. We add the sequence

halt1, halt2, halt3, ...

of independent pronouns and the dependent function pronoun "exits" of which it is understood

```
exits(i) = (if halt1 then res1 else if halt2 then res2 else
                    ... resi) asa halt1 v halt 2 v ... v halti .
```

Here is the table search program

```
valof
  result = exits(2)
       I = 1 fby I+1
   halt1 = tab(I) eq key
    res1 = I
   halt2 = I eq N
    res2 = nil
end
```

which returns i if key is at position i in tab, otherwise nil. And here is the comparison program

```
valof
  result = exits(3)
       I = 1 fby I+1
   halt1 = a(I)<b(I)
    res1 = lt
   halt2 = a(I)>b(I)
    res2 = gt
   halt3 = I eq N
    res3 = eql
end
```

which returns lt, eql or gt according to the lexicographic ordering of the sequences consisting of the first N values of a and b respectively. These pronouns and equations give the programmer more or less the effect of Zahn's "event indicators" (see Zahn [4]) without the cumbersome syntax

5. Reasoning about programs

In verifying or massaging programs which use pronouns there is of course no need to first translate the program into a pronoun free form; one can work directly on the original text using rules such as

$$\text{last}(X+Y) = \text{last}(X) + \text{last}(Y)$$

which involve pronouns.

It is possible to verify programs using transformation rules alone, but it is almost always helpful to be able to use, in addition, inference rules which allow us to deduce assertions about variables. This involves either using clauses as opposed to phrases, or else making precise the notion of making assertions about the locals of a phrase. In either case we find that there are very natural and suggestive rules of inference involving pronouns. Some involve pronouns which are used in proofs only, and not in programs. For example, let us introduce the dependent pronoun "termination" with the implicit definition

$$\text{termination} = \text{eventually}(\text{halt}).$$

Then we have the following rule of inference

$$\text{first } P,\ P \wedge \neg \text{halt} \rightarrow \text{next } P,\ \text{termination} \models \text{last}(P \wedge \text{halt})$$

which is the analog of Hoare's while rule.

6. Extensions

It should be apparent that a great variety of constructs can be simulated using the pronoun method. One great advantage of the method is that the semantics of these constructs can be specified absolutely precisely by an equation or two, without resort to the intricacies of context free grammars, the Vienna Definition Language, or the Scott-Strachey method. A second advantage is that in many cases

the pronoun forms are actually easier to use because the arguments are specified by an unordered set of 'keyword' equations, rather than being specified positionally by plugging expressions in the pigeonholes of some grammatical template. An obvious possibility is to combine the two approaches, eg to define a for loop in which the index value and the range are specified positionally, but in which the final result and error exits are defined with pronouns. Another is to define different phrases within which different equations are assumed to hold; and a third is to allow the user to provide his own implicit equations. This last possibility must not be undertaken lightly, for it amounts to permitting a form of dynamic binding.

References

1. Ashcroft, E.A., and Wadge, W.W., Lucid, a nonprocedural language with iteration, CACM 20, 7 (July 1977), 519-526.

2. Knuth, D.E., Structured programming with goto statements, Comp.Surv. 6, 4 (1974), 261-301.

3. Landin, P.J., The next 700 programming languages, CACM 9, 3 (1966), 157-164.

4. Zahn, C.T., A control statement for natural top down structured programming, Proceedings of the Symposium on Programming Languages, Paris (1974).