A LOGICAL PROGRAMMING LANGUAGE

Ed Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

Research Report CS-79-20
June 1979

(Typed by Mary Wang)

A LOGICAL PROGRAMMING LANGUAGE

Ed Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

Abstract

In this paper we consider a family of languages (USWIM) which is based on Landin's ISWIM (the individual languages being determined by appropriate continuous algebras of data objects and operations on these objects). We give a simple mathematical semantics for USWIM, and also give a system of program manipulation rules and a system of inference rules for reasoning about USWIM programs, the latter using a system of "program annotation" which allows inner, local environments to be discussed. USWIM is the basis on which Structured Lucid is built.

0. INTRODUCTION

It is apparent that the goals of language designers and logicians are quite similar: to develop systems for precisely specifying objects and properties. In both cases this means the study and development of the syntax and semantics of purely formal, as opposed to natural, languages.

It is also apparent that logicians have been eminently more successful. The logicians' languages, such as predicate calculus, are simple, elegant and, above all, semantically well defined. Programming languages, by contrast, are complex, clumsy and, above all, semantically very poorly defined. It is often said that they are "illogical".

Furthermore, the languages of logicians were developed in conjunction with rules of inference, so that reasoning about properties of objects could proceed by simple finite manipulations completely within the language itself. By constrast, formal reasoning about programs, to the extent that it is possible at all, has to be carried out in a separate formal system in which the manipulations are performed on comments on, or a translation of, the original program.

The obvious conclusion is that logic (and mathematics in general) could be usefully applied to the study and design of programming languages. Few would dispute this; but there have always been two points of view about the relationships between logic and computer science.

One point of view sees mathematics as playing primarily a passive role, being used to describe, to model and to classify. The other point of view sees mathematics as playing primarily an active role, being used not so much to describe existing objects as to plan new objects.

These two approaches, which we might call the *descriptive* and *prescriptive* approaches [3], are well illustrated by two important papers by Landin, "A Correspondence between Algol 60 and Church's λ Notation" and "The Next 700 Programming Languages" [6, 7].

In the first Landin defines a translation from Algol 60 into the λ-calculus and so uses logic to describe Algol. In the second he begins with λ-calculus and develops a simple non-procedural language (ISWIM†), with a naturally defined construct (the "where" phrase††) which introduces local variables in a way similar to the Algol block, but which is actually based on the λ-calculus. Landin's first paper represents the descriptive approach and the second represents the prescriptive approach.

Our goal here is to follow Landin's lead in the second paper and develop an uncompromisingly logical language which has "facilities" for functions and scope.

The language, USWIM†††, is in fact a minor variant of ISWIM. Landin, however, gave no direct semantics (ISWIM is a syntactic variant of a subset of the λ-calculus), and neither did he give an inference system for the verification of ISWIM programs. (He did give a system, of

† If you See What I Mean.

†† We are using the term "phrase" rather than Landin's original "clause" because we shall consistently use "phrase" to mean a compound expression or term, and "clause" to mean a compound formula or assertion.

††† U See What It Means.

sorts, for transforming programs, but it is not very useful.) Of course, these omissions are not Landin's fault, because at that time semantics and program verification were in their infancy. We intend to fill in the gaps in Landin's treatment, and the fact that the semantics and transformation and inference rules turn out to be simple, natural and elegant is a vindication of Landin's mathematical approach to language design.

USWIM forms the basis of further development of Lucid [1], as is considered in [2], but we feel that it is also an interesting language in its own right.

1. ISWIM AND USWIM

ISWIM is based on the where phrase, which is an expression qualified by auxiliary definitions. For example

$$\begin{aligned} x^2 + y^2 \quad \texttt{where} \quad & x = a + b \\ & y = a - b \\ \texttt{end} \end{aligned}$$

is a typical where phrase. Such a phrase is a term, i.e. has a value; in the above example, it is the same as that of the term $(a+b)^2 + (a-b)^2$. These constructs can of course be nested, i.e. expressions occurring anywhere in a where phrase may contain where phrases. The variables defined by the equations in the right arm of the where phrase are the locals: their definitions apply only to the right arm and to the left-hand side expression of the phrase. In addition, functions are defined with the formal parameter list on the left hand side, e.g.

$$\begin{aligned} f(x, y) = x^2 - p \cdot q \quad \texttt{where} \quad & p = a \cdot y + b \\ & q = a \cdot x - b \ . \\ \texttt{end} \end{aligned}$$

ISWIM also has a "whererec" phrase in which circular (recursive) definitions are allowed.

The only difference between USWIM and ISWIM is that the former is based on a slightly different phrase, namely the valof phrase. A valof phrase consists of a set of definitions (like those in the right arm of a where phrase) called the body, enclosed by the 'brackets' `valof` (i.e. "value of") and `end` . One of the variables defined in the body must be `result` , whose purpose is to indicate the value of the phrase. Functions are defined as in ISWIM.

The following are legal valof phrases:

```
valof                                valof
      x = a + b                            p = m^2 + 1
      y = a - b                            q = n^2 - 1
  result = x^2 + y^2                   f(c, d) = 1 + valof
end                                                    p = c^2 - d^2
                                                   result = p^2 + 2·p·q
                                                 end
                                        result = f(3, q)
                                     end
```

The most important difference between valof phrases and where phrases is that in the former recursiveness is assumed, i.e. the valof phrase corresponds to the whererec phrase of ISWIM. This eliminates one of the most confusing features of ISWIM, namely the fact that the same variable can refer to different things on the opposite sides of the same equation. Apart from this, there is another reason for preferring valof phrases to where phrases. USWIM is the basis of Structured Lucid [2], and, while it would be possible to base a Structured Lucid on the where phrases of ISWIM, it would then be necessary, as pointed out in [2], to considered the terms E and E where end as having different values (the second is a where phrase with an empty body).

2. SYNTAX OF USWIM

Landin's goal in designing ISWIM was to achieve a clear separation between two aspects of a programming language, the set of given or primitive things and the ways of expressing things in terms of other things. ISWIM is therefore not a single language but a family of languages, each of which is, in Landin's words, "a point chosen from well-mapped space". The coordinate of a point in this well-mapped space is the set of chosen primitives. Since ISWIM and USWIM are based on expressions, to specify the primitives we have to supply a domain of data objects, a collection of operations on these objects, and a collection of symbols used to denote the operations. In other words, a member of the USWIM family is determined by an algebra A ; we will call the corresponding language USWIM(A) . The syntax of USWIM(A) is determined only by the signature of A .

Suppose therefore that we are given an algebra A with signature Σ . That is, Σ is a collection of constant symbols of various arities ("individual constants" being of arity 0). We follow the usual terminology of symbolic logic and refer to the elements of Σ as "constant symbols", even though it is only those of arity 0 which are what computer scientists usually refer to as "constants". This is understandable when we realise that, for example, + , like 3, has the same meaning in all contexts.

We also assume that we have available an unlimited number of variables of various arities (these are what are usually called identifiers). The set of variables is the same for all algebras A .

The nullary variables will also be called "individual variables". Strictly speaking, the constant symbols and variables should be typed to indicate the number of arguments expected, but in practice we will omit these types. Non-nullary constant symbols will often be called operation symbols, and non-nullary variables will often be called function variables.

A USWIM program is simply a term, but to define the class of terms we must also define the classes of definitions and phrases simultaneously and inductively.

A *term* is either

(i) an n-ary constant symbol together with a sequence of n terms as operands (n will be zero if the symbol is an individual constant);

(ii) an n-ary variable together with a sequence of n terms as actual parameters (n will be zero if the variable is an individual variable);

or

(iii) a phrase.

A *phrase* consists of an unordered set of definitions, no two of which have the same definiens , and exactly one of which has has the individual variable result as its definiens.

A *definition* consists of a definiendum which is a term, and of a term consisting of an n-ary variable f (the definiens) together with an ordered set of n distinct individual (nullary) variables (the formal parameters). (f is called a *local* variable of the phrase.)

All variables which are not local variables of a phrase are called *global* variables of the phrase.

This, of course, is an abstract syntax of USWIM (in the sense of MacCarthy [8]), analogous to Landin's abstract syntax for ISWIM. In our examples, (like those already given) we will use a fairly obvious *concrete* linear (or, more realistically, two dimensional) representation in which terms are written using infix notation, definitions are written as equations (with the definiens on the left), and phrases are written as sequences of equations enclosed by the keywords valof and end.† We will not give a precise definition of the concrete syntax. Such a definition would clearly not be particularly complex, but it can not just take the form of a context-free grammar because of the restriction that the formal parameters in a function definition be distinct and the restriction that no variable have two definitions in the same phrase.

In the metalanguage (i.e. when talking about terms in general) we will use expressions like $k(u_0, u_1, \ldots, u_{n-1})$ to denote the term consisting of the n-ary constant k together with the n terms $u_0,u_1,\ldots,u_{n-1}$ as operands. Similarly, we will use expressions like $\mathfrak{f}(u_0, u_1, \ldots, u_{n-1})$ to denote the term consisting of the n-ary variable $\mathfrak{f}$ together with the n terms $u_0,u_1,\ldots,u_{n-1}$ as actual parameters.

† The order of the definitions in a sequence will not be important.

3. SEMANTICS OF USWIM

The algebra A specifies not only the form of USWIM(A) but also its meaning, because A specifies a universe of data objects and assigns to each constant in Σ a meaning, which is an operation over the universe of A. Since USWIM allows arbitrary recursive definitions, we must, in assigning meaning to programs, be able to solve these definitions over the data domain. We therefore assume that the universe of A is a cpo and that the operations of A are continuous,† so that A is what the ADJ [5] school calls a "continuous algebra". The least element of A is $\perp$, which intuitively will be the "result" of non-terminating computations.

Suppose now that we have chosen a signature Σ and a Σ-algebra A.

Even though we have settled upon the meaning of the basic operations we still cannot in general conclude that such and such a term has such and such a value, because the term may have free variables (variables occurring in the term in positions at which there are no relevant definitions). For example, to know the value of the term

```
valof
        $d = a^2 + b^2$
        $e = a^2 - b^2$
    result = valof
                result = $a \cdot q + b$
                    $a = 5 \cdot d$
                    $q = 7 \cdot e$
             end
end
```

† Thus, for example, the "equality operation" must be continuous, and will be denoted by eq to avoid confusion with the "equality relation" = used in definitions.

we must know the values of the variables a and b .

Thus the value of a term depends on both the values of the constant symbols (as represented by the algebra A) and the values of the variables. The latter we represent as an *environment*, by which we mean a function which assigns to each n-ary variable $\mathfrak{f}$ an n-ary function over the universe of A .

These environments are exactly what logicians call "assignments", a name which unfortunately already has other computer science connotations. This coincidence is unfortunate because the word "environment" itself is already widely used by computer scientists to refer to a wide variety of much more complicated structures.

There are two ways to express the fact that the meaning (value) of a term depends on the environment as well as on the algebra. One is to say that given an algebra and an environment, a term has such and such a value; and the second is to say that given an algebra, the value of a term is such-and-such a function from environments to values. In formal logic it is the first approach which is almost always used, whereas in the "Scott/Strachey" semantics it is almost always the second. On this question (as in most others) we choose the approach of the logicians over that of the computer scientists; in this particular instance, because it is notationally and conceptually simpler (avoids higher type objects), and also because it avoids an unnecessary distinction between the algebra and the environment. Of course it is only the 'mathematical' nature of USWIM and the consequent simplicity of the notion of environment which allows us to make the choice.

For a given algebra A, the value of a term t in an environment E is defined inductively as follows:

a) If t is an n-ary constant k together with n arguments $u_0,u_1,\ldots,u_{n-1}$, the value of t in environment E is the result of applying the operation which A associates with k to the values of $u_0,u_1,\ldots,u_{n-1}$ in the environment E.

b) If t is an n-ary variable f together with n actual parameters $u_0, u_1, \ldots, u_{n-1}$, the value of t in the environment E is the result of applying the function associated with f (according to environment E) to the values of $u_0, u_1, \ldots, u_{n-1}$ in the environment E.

c) If t is a phrase the value of t in environment E is the value of result in the least† environment E' agreeing with E (except possibly for the locals of t) which satisfies the definitions in t.

An environment E satisfies a definition if, for all environments E' which differ from E at most in the values associated with the formal parameters of the definition, the value in E' of the left hand side term containing the definiens is the same as the value in E' of the right hand side (definiendum).

This, strictly speaking, is not a definition of the value of t in E since it assumes that least environments exist. However, if the operations in the algebra A are continuous, least environments *do* exist, so the above is a true statement about the value of t in E.

Since a USWIM program is just a term, the above is an informal

† The partial order on the data domain induces an order on the collection of environments.

description of the complete semantics of the language. It is a mathematical semantics rather than an operational one, that is, it does not specify the way one would actually compute the value of a program. If the data domain is reasonably simple (for example, the integers), it is in fact relatively straightforward to specify an operational semantics, using conventional Algol-like recursion-implementation techniques, that agrees with, i.e. realises, the mathematical semantics. In other data domains, especially those where the data objects are infinite, such as that of Lucid [1], the mathematical semantics corresponds to completely different operational concepts like iteration, data flow and coroutines. This subject will be explored in a forthcoming paper.

4. SUBSTITUTION

The semantics of USWIM just given is simple and concise but cannot easily be applied directly in reasoning about programs. Instead, we use the semantics to justify purely syntactic manipulation rules and rules of inference. These rules, because they are purely syntactic (involve only text manipulation) can be used confidently without reference to any semantic or mathematical notions. The rules will be given here without proofs of their validity; such proofs exist, but it would be out of place to include them here, since they tend to be repetitive and lengthy.

In common with all formal logical systems, in order to specify rules of inference we must first define the concepts of free and bound variables, the operation of substitution, and the conditions under which substitution causes no clashes of scope.

The free occurrences of variables in a term are those which refer to objects external to the term itself. The value of a term depends only on the values of the objects referred to by these free variables. Substitution is the operation of replacing such free variables by terms. There are no clashes of scope caused by this operation provided the free variables of the terms being substituted still refer to external objects, i.e. are still free variables.

Informally, a bound occurrence of a variable in a term is an occurrence where the variable is a formal parameter of an enclosing definition or is a local variable of an enclosing phrase. All occurrences in the term which are not bound are said to be free.

More precisely, given any term t and any definition d, the *free occurrences* in t and d are defined as follows:

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then the free occurrences of variables in t are those corresponding to free occurrences in some u_i ;

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then the free occurrences in t are the initial occurrence of f together with occurrences corresponding to free occurrences in some u_i;

(iii) if t is of the form

```
valof
    $e_0$
    $e_1$
    ⋮
    $e_n$
end
```

(where the e_i's are definitions)

then the free occurrences are those corresponding to free occurrences, in some e_i , of variables which are not locals of t ;

(iv) if d is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = a$$

then the free occurrences in d are the initial occurrence of f together with those free occurrences in a of variables other than the formal parameters $x_0, x_1, \ldots, x_{n-1}$ of d .

Any occurrence of a variable in a term which is not a free occurrence is a *bound occurrence*. Any variable which occurs free in a term is a *free variable* of the term.

It is very common to want to talk about the result of replacing all free occurrences of a variable in a term by some other term. Since our variables are function variables, to do this replacement the (substituted) arguments of the variable being replaced must in turn be substituted into the term doing the replacing.

We say that the pair, consisting of the variable $\mathfrak{f}$ to be replaced (and individual variables $x_0, x_1, \ldots, x_{n-1}$ representing its arguments) and the term t to replace it (which usually contains free occurrences of the argument variables), is called an *assignment*. The assignment will be denoted by $\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow t$. A *substitution* is a set of assignments in which the variables to be replaced are all distinct.

We specify the result $t[S]$ of applying the substitution S to the term t in the following informal manner. We work outwards from the innermost terms of t, and whenever we find a free occurrence of a variable $\mathfrak{f}$ to be replaced (say $\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$ occurs in S), if the arguments of this occurrence are $u_0, u_1, \ldots, u_{n-1}$ (after possible substitution) then $\mathfrak{f}$ together with $u_0, u_1, \ldots, u_{n-1}$ will be replaced by a after simultaneously substituting u_i for x_i, $1 \leq i \leq n$.

More precisely, given any term t, any definition d and any substitution S:

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then $t[S]$ is

$$k(u_0[S], u_1[S], \ldots, u_{n-1}[S]) ;$$

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then $t[S]$ is

$$f(u_0[S], u_1[S], \ldots, u_{n-1}[S])$$

unless S has an assignment of the form

$$f(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

in which case $t[S]$ is

$$a[\{x_0 \leftarrow u_0[S], x_1 \leftarrow u_1[S], \ldots, x_{n-1} \leftarrow u_{n-1}[S]\}] .$$

(iii) if t is of the form

```
valof
   $e_0$
   $e_1$
    :
   $e_n$
end
```

(where the e_i's are definitions)

then $t[S]$ is

```
valof
   $e_0[\tilde{S}]$
   $e_1[\tilde{S}]$
    :
   $e_n[\tilde{S}]$
end
```

where $\tilde{S}$ is the result of removing from S all assignments to the locals of t .

(iv) if definition d is of the form

$$g(y_0, y_1, \dots, y_{n-1}) = b$$

then $d[S]$ is

$$g(y_0, y_1, \dots, y_{n-1}) = b[\hat{S}]$$

where $\hat{S}$ is the result of removing from S any assignments to the formal parameters $y_0, y_1, \dots, y_{n-1}$ of d .

Example

In the term

```
x + valof
            z = y + 3
        g(x) = if z > y then z else g(y + x)
      result = h(x + z)
   end
```

there are two free occurrences and three bound occurrences of x , two free occurrences of y , four bound occurrences of z , two bound occurrences of g , one free occurrence of h and one bound occurrence of result .

If we apply the substitution

$$\{x \leftarrow z,\ h(y) \leftarrow g(h(y + 1)),\ y \leftarrow (x + 2)\}$$

the result is

```
z + valof
            z = (x + 2) + 3
        g(x) = if z > x then z else g((x + 2) + x)
      result = g(h((z + z) + 1))
   end.
```

This substitution has several undesirable effects. After substitution, the two free occurrences of x become occurrences of z, but they refer to different objects, since the second is a local variable of the phrase. Similarly the two free occurrences of y become occurrences of "$x + 2$" but again they refer to different objects, since the x in the second is a formal parameter. These are examples of free occurrences of variables in the term doing the replacement becoming bound occurrences in the result of substitution. Another example of this is the substitution for h, where the free occurrence of the variable g becomes bound, since g is a local of the phrase.

We say that a term t *permits* a substitution S if none of these effects occur, i.e., if no free occurrence in a term being substituted becomes a bound occurrence in the result of the substitution.

In the same way, we say that a definition permits a substitution provided the definiendum permits the substitution and no free variables become formal parameters.

More precisely, for any term t, any definition d and any substitution S :

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then t permits S iff each u_i permits S ;

(ii) if t is of the form

$$\delta(u_0, u_1, \ldots, u_{n-1})$$

then t permits S iff each u_i permits S, unless S contains an assignment of the form

$$f(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

in which case it is also required that a permit the substitution

$$\{x_0 \leftarrow u_0[S], x_1 \leftarrow u_1[S], \ldots, x_{n-1} \leftarrow u_{n-1}[S]\}$$

(iii) the term

```
valof
   $e_0$
   $e_1$
   ⋮
   $e_n$
end
```

permits S provided each definition e_i permits the substitution $\tilde{S}$ described earlier, and provided that $\tilde{S}$ has no assignment of the form

$$f(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

with f occurring free in some e_i and some local of t occurring free in a .

(iv) the definition

$$g(y_0, y_1, \ldots, y_{n-1}) = b$$

permits S provided b permits the substitution $\hat{S}$ described earlier and $\hat{S}$ does not contain an assignment of the form

$$f(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

such that $\mathfrak{f}$ occurs free in b and a has a free occurrence of one of the formal parameters $y_0, y_1, \ldots, y_{n-1}$ of d .

5. PROGRAM MANIPULATION RULES

With these technical matters taken care of we can proceed to consider the program manipulation rules.

The rules we present here are all transformation rules; they describe changes which can be made to a term without altering its meaning. If the term has free variables, the value is preserved no matter what value is assigned to those variables. This means that the rules can be used in any context, i.e. can be applied to subterms. As a result, our reasoning about programs can proceed as follows: we begin with the program P_0 and then define a sequence $P_1, P_2, P_3, \ldots$ of derived programs, each P_{i+1} the result of applying a rule to a subterm of P_i , until we arrive at a program P_n which more clearly has some desired property. Since the transformations preserve meaning, P_0 must also have the property. This stepwise or "incremental" approach to reasoning about programs closely resembles the approach mathematicians use in trying to prove something about, say, a system of differential equations. Mathematicians do not simply derive a series of assertions about the system; they transform the system itself, by adding new variables, eliminating old ones, integrating parts into closed forms, expanding power series, and so on. In logic itself most of the proof systems (e.g. the natural deduction systems) proceed by taking assertions apart and putting them together, but there is one system (Craig's Linear Reasoning [4]) which is based on a transformational approach. In Craig's system one proves an implication by transforming the hypothesis into the conclusion.

Another important characteristic of proofs in our system is that they are modular or 'nested' in that they follow the scope structure of the term to which they are applied. By this we mean that each application of a rule involves only the body of a single phrase. (To apply such rules we often have to use a rule for bringing definitions into a phrase.) This means that one can concentrate one's attention on one part of the program at a time, transforming its equations without altering, or even taking into account, definitions in inner or enclosing phrases.

The first of our rules, the *import rule* , captures the idea that the definition of a variable applies to all free occurrences of the variable inside inner phrases, i.e. the idea that scope propagates inward. It says that we can add to any phrase a definition occurring (at the top level) in the smallest enclosing phrase provided the variable defined is not a local of the inner phrase, and is not a formal parameter of the definition in whose right hand side the inner phrase occurs. For example, the rule allows us to transform

```
valof                                      valof
      a = p + q                                    a = p + q
      b = p - q                                    b = p - q
   result = valof                              result = valof
                 b = a + 1        into                       a = p + q
             result = b^2                                    b = a + 1
          end                                           result = b^2
end                                                     end
                                           end
```

The *calling rule* expresses the idea that the definitions in a phrase really are equations, i.e. that the right hand side is really equal to the left hand side. It says that if a phrase contains the definition

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) = a$$

then the substitution $\{\mathfrak{f}(x_0,x_1,\ldots,x_{n-1}) \leftarrow a\}$ can be applied to any definition in the phrase, provided the subsitution is permitted, that is, the free variables in a remain free variables of the definition after substitution. For example, the rule allows us to transform

```
valof                                  valof
    f(x) = x^2 - y^2                       f(x) = x^2 - y^2
                            into
     y = p + 3                              y = p + 3
  result = f(3 + y) - f(p)             result = ((3 + y)^2 - y^2) - f(p)
end                                    end
```

Because of the way in which substitution works, this rule also captures the idea that the definitions with formal parameters are definitions of functions.

The *local renaming rule* expresses the idea that the local variables of a phrase are just dummy variables, that the actual variables chosen are not important, and that the enclosing phrases are 'shielded' from the definitions in the phrase. It says that in a phrase whose locals (other than result) are $v_0, v_1, v_2, \ldots, v_{m-1}$ we can choose any other sequence $w_0, w_1, \ldots, w_{m-1}$ of distinct variables (of corresponding arities) and replace all free occurrences of $v_0, v_1, \ldots, v_{m-1}$ in the definitions of the phrase by $w_0, w_1, \ldots, w_{m-1}$, provided the substitutions are permitted, i.e. as long as no v_i occurs where the corresponding w_i would be bound, and provided no w_i is a free variable of the original phrase. For example the rule allows us to transform

```
valof
        a = 3 - q(1, 1, 5)
      f(x) = x + valof
                           g(x) = x^2 - a
                             c = x^2
                        result = g(f(x)) + c
                 end
   q(r, s, t) = r^2 - a•r•s + f(t)
      result = a
end
```

into

```
valof
            $b = 3 - k(1, 1, 5)$
        $v(x) = x +$ valof
                                $g(x) = x^2 - b$
                                   $c = x^2$
                              result $= g(v(x)) + c$
                    end
    $k(r, s, t) = r^2 - b \cdot r \cdot s + v(t)$
        result $= b$
end
```

but we could not change f to g because the occurrence of f in the inner phrase is in a context in which g is bound. Also we can not rename the locals of the inner phrase to change c to a because the existing free occurrence of a in the inner phrase would become bound.

There is also a similar renaming rule for formal parameters, which expresses the fact that they too are dummy variables. (The *formal parameter renaming rule* .)

The *amalgamation rule* is so-called because it allows us to amalgamate a term involving several phrases into a single phrase in which result is defined to be an analogous term, for example to transform the sum of two phrases into a phrase in which result is defined as a sum.

Suppose then that e is any expression, that $v_0, v_1, \ldots, v_{n-1}$ is a sequence of individual variables and that

$p_0, p_1, \ldots, p_{n-1}$ is a sequence of phrases having (other than result) no locals in common, and none of whose locals is a v_i or occurs free in e . We define the phrase q as that formed by taking the definition result = e together with, for each i , all definitions in p_i with result replaced by v_i . Then the term

$$e[\{v_i \leftarrow p_i \mid i < n\}]$$

(the term involving the phrases) may be replaced in any context by q provided all the substitutions indicated are permitted.

For example, suppose that the expression e is $a - b \cdot z$, that n is 2 and v_0 and v_1 are a and b respectively and that p_0 and p_1 are

```
valof                                valof

          x = m + n                            r = m + k

          y = m•n         and                  s = m - k

      result = x² - y                     result = r•s + 5

end                                  end
```

Then the expression

```
  / valof           \     / valof              \
 |                   |   |                      |
 |      x = m + n    |   |        r = m + k     |
 |                   | - |                      | • z
 |      y = m•n      |   |        s = m - k     |
 |                   |   |                      |
 |  result = x² - y² |   |   result = r•s + 5   |
 |                   |   |                      |
  \ end             /     \ end                /
```

can always be replaced by

```
valof
        $x = m + n$
        $y = m \cdot n$
        $a = x^2 - y$
        $r = m + k$
        $s = m - k$
        $b = r \cdot s + 5$
   result = $a - b \cdot z$
end.
```

An interesting special case of this rule is obtained when $n = 1$ and v_0 does not occur free in e. The inverse of this rule[†], which in this case we shall call the *result rule*, tells us that we can replace any phrase of the form

```
valof
   ⋮
   result = $e$
   ⋮
end
```

by e itself (the conditions of the general rule require that no local of the phrase occur free in e).

We mentioned that mathematicians, in the course of manipulating a set of differential equations, might introduce new variables and

† In fact all the rules can be applied in either direction. This means that from a term t we can obtain a term s by the "inverse of rule r" provided that rule r could be applied to term s to yield term t.

discard old ones. There is exactly such a rule for USWIM. The *addition rule* allows us to add to a phrase *any* definition whatsoever provided only that the variable being defined does not occur free in the phrase, and is not already a local variable.

The inverse of the addition rule, called the *deletion rule*, allows us to eliminate a definition in the same circumstances. For example, the addition rule allows us to transform

```
valof
    f(x, y) = x^2 + a•x•y + y^2
        b = 3 - a
     result = f(2, b)
end
```

into

```
valof
    h(x, y) = x^2 + y^2 - f(x, y)
    f(x, y) = x^2 + a•x•y + y^2
        b = 3 - a
     result = f(2, b)
end
```

and the deletion rule allow us to transform

```
valof
        a = 3
        b = 4
    result = x^2 - b•x
end
```

into

```
valof
        b = 4
    result = x^2 - b•x
end
```

Finally, there is a rule, the *basis rule*, which reflects the fact that USWIM is, in a sense, an extension of an algebraic language. It says that if the equation

$$t_1 = t_2$$

is true in the algebra A (here t_1 and t_2 are simple terms without phrases and the equation is considered to hold for all values of the free variables) then in any term *any* occurrence of t_1 may be replaced by t_2. This allows us to carry over the laws of the data domain so that, for example,

```
valof                                         valof
    f(a, b) = (a + b)^2                           f(a, b) = a^2 + 2·a·b + b^2
                              can become
     result = f(3, 5)                              result = f(3, 5)
end                                           end
```

if the algebra A consists of the integers with the usual operations, in which the equation

$$(a + b)^2 = a^2 + 2 \cdot a \cdot b + b^2$$

holds for all a and b.

6. BINDING AND CALLING

If the language and rules of inference of the underlying algebra A are fairly standard, for example dealing with numerical quantities and interpreting the operation symbols in the usual way, then the rules of inference of USWIM just given are sufficient to successively transform any program (without free variables), whose value is defined, into a term without phrases denoting that value. This essentially corresponds to 'executing' the program.

For example, consider the program

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = f(3)
end.
```

Applying the calling rule we get

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = if 3 ≤ 0 then 1 else 3•f(3 - 2)
end.
```

Now applying the basis rule for properties of arithmetic we get

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = 3•f(2)
end.
```

By repeating this process three more times we will get

```
valof
        $f(n)$ = if $n \le 0$ then 1 else $n \cdot f(n - 1)$
    result = 6
end.
```

Now applying the result rule, this term becomes simply 6.

The rules can also be used to answer questions about function-calling that in conventional programming languages are thought of operationally and are settled by conscious design decisions. These are the questions of the type of "parameter passing mechanism" used and whether the "binding" of global variables of functions is static or dynamic.

It is easy to see that the "parameter passing mechanism" is *not* call-by-value by considering the following program

```
valof
    $f(x, y)$ = if $x \le 0$ then 0 else $f(x-1, f(x, y))$
     result = $f(1, 0)$
end.
```

If the calling "mechanism" were call-by-value, the value of this program would be $\perp$, since evaluation of $f(1, 0)$ in turn requires evaluation of $f(1, 0)$. However, it is easy to see that the rules allow us to generate the following sequence of equivalent programs:

```
valof
    f(x, y) = if x ≤ 0 then 0 else f(x-1, f(x, y))
     result = f(0, f(1, 0))
end
```

(by the calling and basis rules)

```
valof
    f(x, y) = if x ≤ 0 then 0 else f(x-1, f(x, y))
     result = if 0 ≤ 0 then 0 else f(0-1, f(0, f(1, 0)))
end
```

(by the calling rule)

```
valof
     result = 0
end
```

(by the basis rule and the deletion rule)

```
0
```

(by the result rule).

Thus, the value of the program is 0 , not $\perp$.

As for the "binding" used, we can consider the following program:

```
valof
          $a$ = 1
       $f(x)$ = $x + a$
       result = valof
                          $a$ = 2
                     result = $f(2)$
                  end
end.
```

If the binding were dynamic, we would expect the value of this program to be 4, since it would be the inner definition of a that would be used by the "call" of $f(2)$. There are two ways to see that this is not the case. First, by the local renaming rule, we can rename the variable a either in the outer phrase or in the inner phrase, and in both cases the global variable of f will be the local defined in the outer phrase. Second, we can actually "execute" the program. This involves applying the calling rule to $f(2)$. However, the outer definition of result (which is the one, containing $f(2)$, which is in the phrase containing the definition of f) does not permit the substitution $f(x) \leftarrow (x + a)$ because the variable a , which is free in "$x + a$" , becomes bound in the resulting definition. Therefore we must first apply the local renaming rule, either to the inner or outer phrase, so that the substitution (which will have been changed if we renamed a in the outer phrase) is then permitted. It is then straightforward to see that the program reduces to 3.

Thus USWIM uses "static binding", as in fact all languages do which allow renaming of local variables.

7. INFERENCE RULES

The USWIM rules which we have described cannot by themselves be used to prove *assertions* about programs because these rules apply only to programs; they constitute a calculus of program transformations rather than a logic of programs. If we want to manipulate assertions about programs as well, we must extend our formal system by adding some class of assertions together with a collection of rules for deriving assertions. One way to do this is to take some existing logical system which already deals with terms and 'embed' USWIM in it by expanding the class of terms to include USWIM programs.

The most obvious choice for such a system is the first order language L whose individual variables are the nullary USWIM variables and whose operation symbols are taken from Σ . We generalize L to the language L^* by allowing a USWIM phrase whose free variables are all nullary to appear in a formula anywhere an ordinary term is permitted. In this 'extended' logic, assertions about programs are assertions about the values of programs and are just formulas in which the program appears; for example,

$$\forall i \left(\left(\begin{array}{l} \texttt{valof} \\ \qquad sq(n) = n \cdot n \\ \qquad \texttt{result} = sq(i-1) \cdot sq(i+1) \\ \texttt{end} \end{array} \right) = i^4 - 2 \cdot i^2 + 1 \right) .$$

There is no difficult in using the mathematical semantics of USWIM given earlier to extend the semantics of L , i.e. to define what it means for an L^*-formula to be true in a given environment.

Program-equivalence assertions are just equations between programs with their (common) input (free) variables universally quantified; for example,

$$\forall n \forall r$$

$$\left(\begin{array}{l} \texttt{valof} \\ \quad fac(x) = \texttt{if } x < 1 \texttt{ then } 1 \\ \qquad\qquad\quad \texttt{else } x \cdot fac(x-1) \\ \quad \texttt{result} = fac(n)/(fac(r) \cdot fac(n-r)) \\ \texttt{end} \end{array} \right) = \left(\begin{array}{l} \texttt{valof} \\ \quad h(x, y) = \texttt{if } x > y \texttt{ then } 1 \\ \qquad\qquad\quad \texttt{else } x \cdot h(x+1, y) \\ \quad \texttt{result} = h(n-r+1, n)/h(1, r) \\ \texttt{end} \end{array} \right)$$

There is, however, a serious problem with this approach, and that is that while the operation symbols of A can be immediately incorporated into L^* , the relation symbols cannot, they are just operations in A which happen to yield what A's interpretation of if...then...else regards as truth values. In fact, since Σ is a one-sorted signature it does not even make sense to talk about "relation symbols" in Σ .

We can, of course, define a many-sorted version of USWIM whose parameter would be a many-sorted algebra, and we could require that bool be one of these sorts and that A interpret it in some standard way. Nevertheless we could still not use operations in Σ of (result) type bool as relations in L^* because the value of such operations could be the 'undefined' element $\perp$ of A. This is possible because A is required to be a continuous algebra - and we cannot drop this requirement because our definition of the meaning of a USWIM phrase requires the existence of fixed-points of arbitrary recursive definitions.

There are two solutions to this problem. One is to distinguish completely between tests used in a program and actual relations over the universe of A (this involves using two different forms for every basic test in A, e.g. distinguishing between $<$ and less-than, $>$ and greater-than, $\geq$ and greater-than-or-equal in the same way as between $=$ and eq). An alternate solution is to adopt a three-valued logical system plus additional rules specifying circumstances under which conventional two-valued reasoning may be used. The first method is formally simple but in practice is very cumbersome, whereas the latter requires a more elaborate formal specification but is much easier to use in practice. Here we shall use the latter method.

Because the use of full predicate logic involves complications which we would rather avoid here, we will instead use as the basis of our assertion language the equational algebraic system of signature Σ. We extend it by allowing, as generalized assertions, universally quantified equations between USWIM terms. More precisely, our assertions are of the form

$$\forall x_0\ \forall x_1\ \dots\ \forall x_{n-1}(t_1 = t_2)$$

for some natural number n , some sequence $x_0, x_1, \dots, x_{n-1}$ of (for simplicity) individual variables, and some terms t_1 and t_2 . The above assertion e is true in E (in symbols $\models_E e$) iff the value of t_1 in E' is the value of t_2 in E' for any environment E' differing at most from E in the values E' gives to some of the x_i . The free variables of e are those variables other than the x_i which are free in t_1 or t_2 . The assertion e permits a substitution S iff both t_1 and t_2 permit the substitution $\hat{S}$ obtained by removing from S all assignments to any x_i , provided no x_i occurs free in any assignment in $\hat{S}$, and provided both t_1 and t_2 permit $\hat{S}$. When the substitution is permitted the result $e[S]$ is

$$\forall x_0\ \forall x_1\ \dots\ \forall x_{n-1}(t_1[\hat{S}] = t_2[\hat{S}]) \quad .$$

In future we shall refer to such universally quantified equations simply as "equations", and equations without quantifiers will be "basic equations".

In proving equations from other equations we can use the equational rules of substitution and replacement. More precisely,

(i) *Substitution*:

from any equation of the form

$$\forall x_0\ \forall x_1\ \dots\ \forall x_{k-1} d \quad ,$$

with d any basic equation, infer the equation

$$\forall y_0 \ \forall y_1 \ \ldots \ \forall y_{m-1} d[S]$$

for any permitted substitution S of the form

$$\{x_0 \leftarrow u_0, \ x_1 \leftarrow u_1, \ldots, \ x_{k-1} \leftarrow u_{k-1}\}$$

and $y_0, y_1, \ldots, y_{n-1}$ any variables not free in the original equation;

(ii) *Replacement*:

from an equation of the form

$$\forall x_0 \ \forall x_1 \ \ldots \ \forall x_{n-1} (t_0[v \leftarrow a] = t_1[v \leftarrow a])$$

and an equation

$$\forall y_0 \ \forall y_1 \ \ldots \ \forall y_{m-1} (a = b)$$

infer the equation

$$\forall x_0 \ \forall x_1 \ \ldots \ \forall x_{n-1} (t_0[v \leftarrow b] = t_1[v \leftarrow b])$$

provided the substitutions are permitted and any x_i appearing free in a or b is among the y_i .

Since we will be using other rules of inference as well, we need an extra *generalization rule* which allows us to infer

$$\forall y_0 \ \forall y_1 \ \ldots \ \forall y_{m-1} d$$

from d whenever no y_i occurred free in the assumptions from which d was derived. (Here d need not be a basic equation.)

These are exactly the ordinary equational rules except that (i) the terms may contain USWIM phrases, and (ii) quantification is explicit. In ordinary equational algebra all variables are implicitly universally quantified, but we have made quantification explicit in order to avoid a distinction between variables in programs and variables in assertions. We therefore need a third rule (which we will not formulate precisely) allowing us to permute the order in which the variables are quantified in an equation.

These rules are sound in the following sense: if we can, using these rules, derive an equation e from a set $d_0, d_1, \ldots, d_{n-1}$ of equations, then e is true in every E in which all the d_i are true; in symbols,

$$d_0, d_1, \ldots, d_{n-1} \models e \quad .$$

Our program transformation system allows an extra rule which express precisely the fact that transformed programs are equivalent. If our transformation rules allows us to transform the term t_1 into the term t_2 we may infer the equation

$$\forall x_0 \ \forall x_1 \ \ldots \ \forall x_{n-1} (t_1 = t_2)$$

for any individual variables $x_0, x_1, \ldots, x_{n-1}$.

The substitution and replacement rules are general because they are sound no matter what the algebra A is. In studying a particular instance of the USWIM family, however, it is possible to use particular but very useful rules of inference valid only with reference to the particular algebra. An important example is the mathematical induction rule of the algebra N of natural numbers (with $\Omega \in \Sigma$ and $N(\Omega) = \perp$). To prove an equation $\forall v \ e$ from a set of assumptions it is enough to prove the equations

$$\mathcal{C}[\{\upsilon \leftarrow \Omega\}] \qquad \text{and} \qquad \mathcal{C}[\{\upsilon \leftarrow 0\}]$$

from the assumptions, and also prove $\mathcal{C}[\{\upsilon \leftarrow \upsilon+1\}]$ from the set of assumptions with $\mathcal{C}$ added (as the induction hypothesis), provided υ is not free in any assumption in the set.

The problem that remains with the "embedding" approach just described is that assertions about programs are just equations in which the programs appear as terms, so that they are, in fact, assertions about the *values* of programs. Programs are therefore treated as black boxes whose input-output activity is all that can be discussed. Now it is certainly true that, in the end, the goal of our reasoning is a statement about the meaning of a whole program, i.e. about its value; but in the course of working towards that goal we might want to make assertions about parts of the program. For example in showing that

```
valof
          d = $a_1 \cdot b_2 - a_2 \cdot b_1$
          e = $c_1 \cdot b_2 - b_2 \cdot c_1$
     result = $e/d$
end
```

has a certain value we would naturally want to prove that (say) $d = k+1$ (and so is positive) inside the phrase. Since this is a statement about the phrase's internal environment, we cannot do this literally, with the 'embedding' system. In such a system,the formalisation of reasoning about the interior of a program involves taking the program apart and putting it back together again, just as described in Section 5.

Clearly, what is required is some system which allows modular, local reasoning *about* a program in the same way that the transformation rules given in Section 5 allow modular, local transformations *of* a program. In reasoning about programs or in documenting them programmers often "annotate" them with comments (written, say, to the right of the text) which refer to the 'local' values of program variables. The system we propose is simply a formalisation of this idea.

We first define two classes of syntactic objects, which we respectively call *annotated terms* and *annotated definitions*, as follows:

An *annotated term* is either

(i) a constant in Σ together with an appropriate number of operands, each of which is an annotated term;

(ii) a variable together with an appropriate number of actual parameters, each of which is an annotated term;

(iii) an annotated phrase, i.e. a set of annotated definitions, one of which has **result** as its definiens and no two of which have the same definiens, *together with* a set of universally quantified equations. (This set of equations is called the *paraphrase* of the phrase, and its elements are called *annotations*.)

An *annotated definition* is similar to a definition (with the definiens and definiendum defined accordingly) except that the definiendum is an annotated term.

We will extend our concrete two-dimensional representation of terms by writing annotations to the right of the phrase to which they are attached, with a vertical line between:

```
valof
        $b = 3 \cdot k$               |      $k = j+1$
        $c = 2 \cdot k^2$             |
    result = valof
                  $d = b^2 - 4 \cdot c$                 |    $b^2 = 9 \cdot k$
              result = $(-b + sqrt(d))/2$     |    $d = k^2$
          end
    end.
```

An annotated phrase has two 'values', its value as a term (a 'data' value), and its value as an assertion (a truth value). The data value is simply the value of the term which results when the annotations are 'thrown away',and the truth value is the conjunction of the annotations in the phrase, each considered as referring to the local environment of the phrase to which it is attached. The idea that annotations refer to local environments can be made precise as follows. Given any annotated term t , any annotated definition d and any environment E :

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then t is true in E iff each u_i is true in E ;

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then t is true in E iff each u_i is true in E ;

(iii) if t is of the form

```
valof
    e_0    |   p_0
    e_2    |   p_2
     :     |    :
    e_n    |   p_{m-1}
end
```

let E' be the least environment which agrees with E (except possibly for the locals of t) which satisfies the definitions obtained by de-annotating the e_i ; then t is true in E iff each p_j is true in E' and each e_i is true in E' ;

(iv) if annotated definition d is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = a$$

then d is true in E iff a is true in every environment E' which agrees with E except possibly for the values given to the formal parameters $x_0, x_1, \ldots, x_{n-1}$ of d .

In future, we will drop the word "annotated", and when we want to indicate the part of a phrase, term etc. without its annotation, we will use the word "de-annotated".

Now, if programs are annotated, we cannot use the program manipulation rules of Section 5 without first modifying them to take account of annotations. We will not go through all these modifications here, because they are all rather obvious. The general consideration is that the free variables of a phrase now include the free variables of the annotation that are not locals of the phrase, and no changes to the de-annotated phrase can be allowed which change the meanings of the annotations. This usually means that the changes made to a de-annotated phrase must also be made to its paraphrase (as, for example, in the local renaming rule).

As well as the program manipulation rules, we now also have rules that deal explicitly with annotations†.

The *discard rule* allows us to remove any annotation from any paraphrase.

The *consequence rule* allows us to add to any paraphrase any equation which is an A-consequence of the annotations already in the paraphrase. In other words, if

$$d_0, d_1, \ldots, d_{n-1} \models e$$

and each d_i is in a particular paraphrase, then e may be added to the paraphrase. In particular, if e can be obtained from the d_i using general or particular rules of inference, it may be added to the paraphrase.

The *definition rule* allows us to add to a paraphrase the equation

$$\forall x_0 \forall x_1 \ldots \forall x_{n-1}(\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) = a)$$

whenever the phrase contains the definition

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) = a \ .$$

(Alternatively, we could allow the calling rule to be applied to annotations as well as to de-annotated phrases.)

† These inference rules, unlike the program manipulation rules, are not invertible.

The *import rule* allows us to add to the paraphrase of an inner phrase any equation in the paraphrase of the immediately enclosing phrase provided it has no free occurrences of variables local to the inner phrase, or of the formal parameters of the definition containing the inner phrase. (This rule is simply a version, for annotations, of the import rule for program manipulation, so using the same name should not cause any confusion.)

We can also generalise the amalgamation rule, so that we can replace q by $e[\{v_i \leftarrow p_i \mid i < n\}]$ (or vice versa) in annotations as well as de-annotated terms.

The *export rule* says that if we have annotation e in a phrase p whose enclosing definition has formal parameters $x_0, x_1, \ldots, x_{n-1}$, and the free variables of e are not locals of the phrase (other than, possibly, $\mathtt{result}$), then we can add to the paraphrase of the enclosing phrase the annotation

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}\ e[\{\mathtt{result} \leftarrow p\}]\ .$$

The rules are sound in this sense: if an annotated term t' can be obtained from an annotated term t, then t' is true in any environment in which t is true.

The rules just described for annotated terms all have the property that their use involves only a *one-way* flow of information *from* the de-annotated program *to* the annotations. If t' can be obtained from t using the rules of inference and the manipulation rules, the corresponding de-annotated terms will have the same value, but the de-annotated version of t' could have been obtained from the de-annotated version of t using just the program manipulation rules. The annotation rules described so far are useful for verification but not for program transformation.

There is a rule which allows annotations to be used to change the de-annotated program. The *modification rule* says (in its simplest form) that if a paraphrase contains the basic equation $a = b$, then certain occurrences of term a in definienda in the phrase may be replaced by term b. This simplest form of the rule would not allow us to substitute b for a in contexts in which some of the free variables of a or b are formal parameters of a definition in the phrase (it clearly would be incorrect to do so, since the assertion $a = b$ can not be referring to the formal parameters). A more general form of the rule allows us to substitute b for certain occurrences of a in this situation if the paraphrase contains

$$\forall x_0, \forall x_1, \ldots, \forall x_{n-1}\ a = b$$

where $x_0, x_1, \ldots, x_{n-1}$ are the formal parameters of the definition in question.

The restrictions on the occurrences of a are not just those necessary to avoid a clash of variables; it is also necessary to take the dependencies of the various locals of the phrase into account, in order to avoid perturbing the least local environment of the phrase.

To understand the necessity of the restriction, consider a program in USWIM(N) containing the definition

$$f(n) = \text{if } n < 1 \text{ then } 1 \text{ else } n \cdot f(n-1)$$

we might deduce the true annotation

$$\forall n(\text{if } n < 1 \text{ then } 1 \text{ else } n \cdot f(n-1) = (n + (1-n)) \cdot f(n-1))$$

(recall that e.g. $1 - 3 = 0$ in N), and if we could apply this to the program using the more general form of the modification rule, the definition of f would become

$$f(n) = (n + (1-n)) \cdot f(n-1) \quad .$$

The problem now is that the factorial function is no longer the *least* solution of this definition; the least solution is the function whose value is always $\perp$. Many and subtle examples of this phenomenon could be given.

We see then that we can change the meaning of the program by substituting equals for equals, because the new definition may have *less* information than the old one. One way to ensure the correctness of a substitution is to require that the equation to be applied is a substitution instance of a more general equation, no free variable of which depends (in the phrase) on the definiens of the definition to which it is applied.

More precisely, suppose that p is an annotated phrase containing a definition of the form

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) = a[\{v \leftarrow t_1\}]$$

(where the substitution is permissible) and that the equation

$$\forall x_0\ \forall x_1\ \dots\ \forall x_{n-1}(t_1 = t_2)$$

can be obtained from an annotation e in the paraphrase of p by one application of the substitution rule. Then the definition may be changed to

$$f(x_0, x_1, \dots, x_{n-1}) = a[\{v \leftarrow t_2\}]$$

provided the substitution in permissible and there is no free variable w of e such that $\langle w, f\rangle$ is in the transitive closure of the relation

$$\{\langle y, z\rangle \mid y \text{ and } z \text{ are locals of } p \text{ and } z \text{ occurs free in the definition of } y\} .$$

The requirement essentially implies that the definition being changed could not usefully have been used to derive the equation being used.

This requirement prevents the erroneous substitution mentioned earlier because f occurs free in the equation and $\langle f, f\rangle$ is in the transitive closure of the relation for the phrase. On the other hand, the equation

$$\forall n(n \cdot f(n-1) = f(n-1) \cdot n)$$

can be applied if the annotation

$$\forall x \forall n(n \cdot x = x \cdot n)$$

is first attached to the phrase.

We will illustrate our transformation and manipulation rules by showing that the following inefficient USWIM(N) program to compute the sum of the first n squares,

```
valof
      sum(m) = valof
                     sq(k) = if k < 1 then 0 else sq(k-1) + (2•k-1)
                         b = sq(m) + sum(m-1)
                    result = if m < 1 then 0 else b
                end
      result = sum(n)
end,
```

is equivalent to the term $n \cdot (n+1) \cdot (2 \cdot n+1) \div 6$.

The first step is to use the definition rule to add the equation

$$\forall k (sq(k) = \text{if } k < 1 \text{ then } 0 \text{ else } sq(k-1) + (2 \cdot k-1))$$

as an annotation to the inner phrase. Next we use mathematical induction to prove

$$\forall i (sq(i) = i^2)$$

by induction on i .

The 'base' steps, namely proving $sq(0) = 0^2$ and $sq(\Omega) = \Omega^2$, are straightforward, and the induction step involves proving

$$sq(i+1) = (i+1)^2$$

from the assumption $sq(i) = i^2$ (notice that i does not occur free in our other assumptions). To do this we substitute into the equation derived from the definition of sq , giving

$$sq(i+1) = \texttt{if } (i+1) < 1 \texttt{ then } 0 \texttt{ else } sq((i+1) - 1) + (2 \cdot (i+1) - 1) \quad .$$

Replacements using equations like

$$\forall i((i+1) - 1 = i)$$

give us

$$sq(i+1) = \texttt{if } (i+1) < 1 \texttt{ then } 0 \texttt{ else } sq(i) + 2 \cdot i+1 \ .$$

Replacing $sq(i)$ by i^2 (i.e., by using the induction hypothesis) we get

$$sq(i+1) = \texttt{if } (i+1) < 1 \texttt{ then } 0 \texttt{ else } i^2+2 \cdot i+1$$

and since

$$(i+1)^2 = \texttt{if } (i+1) < 1 \texttt{ then } 0 \texttt{ else } i^2+2 \cdot i+1$$

is true in N , we can replace and get

$$sq(i+1) = (i+1)^2$$

as required.

The induction rule therefore allows us to add $\forall i \ sq(i) = i^2$ to our annotations, and substitution gives $sq(m) = m^2$ and since m is not a local of the phrase and sq is not defined in the phrase directly or indirectly in terms of b , we can use the modification rule to replace the occurrence of $sq(m)$ by m^2 in the definition of b . Then we use the calling rule on the occurrence of b in the definition of `result` giving

```
valof
    sum(m) = valof
                 sq(k) = if k < 1 then 0 else sq(k-1) + (2•k-1)
                     b = m^2 + sum(m-1)
                result = if m < 1 then 0 else m^2 + sum(m-1)
             end
    result = sum(n)
end
```

(after discarding the annotations). In the inner phrase, *sq* and *b* occur free at most in their own definitions; these definitions may therefore be deleted, giving

```
valof
    sum(m) = valof
                result = if m < 1 then 0 else m^2 + sum(m-1)
             end
    result = sum(m)
end.
```

The next step is to use the result rule and replace the entire inner phrase by the definiendum of result , yielding

```
valof
    sum(m) = if m < 0 then 0 else m^2 + sum(m-1)
    result = sum(n)
end.
```

We can now use annotations much as we did before to prove the equation

$$\forall m(sum(m) = m\cdot(m+1)\cdot(2\cdot m+1) \div 6)$$

by induction on m , then use substitution and modification to get

```
valof
    $sum(m)$ = if $m < 0$ then 0 else $m^2 + sum(m-1)$
    result = $n \cdot (n+1)\cdot(2\cdot n+1) \div 6$
end.
```

Discarding the definition of *sum* and using the result rule finally gives the term $n\cdot(n+1)\cdot(2\cdot n+1) \div 6$. Since we have transformed the original program p into this term, and since we did not use any assumptions about n , we conclude

$$\forall n(p = (n\cdot(n+1)\cdot(2\cdot n+1) \div 6)$$

is true in N .

The USWIM rules do not in themselves make verifying programs any easier mathematically (although they make it easier notationally). The real significance of the rules is that they allow the proofs derived to be completely precise, i.e. broken down into a series of small steps each of which is the application of a simple rule. Naturally this degree of precision would be possible only with the aid of a mechanical proof checker capable of 'interpolating' simple steps. Such a checker/verifier based on USWIM would be no more complicated than many existing systems, and would allow a user to perform sophisticated manipulation with complete confidence.

8. REASONING BY SCOTT INDUCTION

We have mentioned that the definitions in a phrase can be thought of as true assertions about the locals, and two of the manipulation rules given can (as we pointed out) be considered as justifications of this way of thinking. Neither of these rules, however, make use of the fact that the environment inside a phrase is the *least* one which makes the equations true. The two rules given express the fact that the values of the locals are a fixed-point of the equations, but they do not express the fact that they are the *least* fixed point.

This is not as serious a deficiency as it might seem, because in very many cases the fixed-point is unique anyway. Sometimes, however, there is more than one fixed-point, and,even when there isn't, it is often easier to derive a particular result using minimality even when minimality is not strictly necessary.

Proof rules which can be used to derive assertions about the least (but not necessarily all) fixed points of a continuous function are known as fixed-point induction rules, and several different ones are known. We will describe a USWIM rule which is based on the "Scott Induction" principle which has the useful property that it can be stated without explicit reference to the approximation relation in the domain in question.

The Scott rule is the following: given any cpo, any continuous function τ over the cpo with least fixed point κ , and any property p : to show $p(\kappa)$ show $p(\bot)$ and then show that $p(\alpha)$ implies $p(\tau(\alpha))$ for any α in the cpo. The rule is valid provided that p is "admissible" in the sense that the lub of a chain of elements in the cpo possesses the property whenever every element in the chain does.

The rule itself is a meta-rule, i.e. it refers to semantic objects. The corresponding USWIM rule is an object rule in that it refers to syntactic objects and gives conditions under which annotations can be added to terms. Suppose then that t is a phrase, that $v_0, v_1, \ldots, v_{n-1}$ are any of the locals of the phrase, and that q is some universally quantified equation about the v_i and other variables which we wish to prove by induction on the v_i. (q is bound to be admissible.) We assume that Σ has nullary symbol Ω which A interprets as $\perp$, and that $w_0, w_1, \ldots, w_{n-1}$ are variables which are not local to the phrase and do not occur free in any annotation in the phrase. Then in order to justify adding q to the phrase we must

(i) be able to add q to the paraphrase of the phrase t' formed by changing the definition of each v_i to

$$v_i(x_0, x_1, \ldots, x_{r_i-1}) = \Omega$$

(where r_i is the arity of v_i);

(ii) be able to add q to the paraphrase of the phrase t'' which results from applying the substitution

$$\{v_i(x_0, x_1, \ldots, x_{r_i}) \leftarrow w_i(x_0, x_1, \ldots, x_{r_i}) \mid i < n\}$$

to the definienda of the v_i, and adding as an annotation the result of applying the above substitution to q.

As an example of the application of the rule, suppose that we wish to add the annotation

$$\forall n\ sum(n) = fib(n+2) - 1$$

to the phrase

```
valof
    fib(n) = if n < 2 then 1 else fib(n-1) + fib(n-2)
    sum(n) = if n < 0 then 0 else fib(n) + sum(n-1)
    result = sum(20)
end.
```

We must first show that we can add the given annotation to the phrase

```
valof
    fib(n) = Ω
    sum(n) = Ω
    result = sum(20)
end;
```

and then show that the same annotation can be added to the annotated phrase

```
valof
    fib(n) = if n < 2 then 1 else a(n-1) + a(n-2)  | ∀n b(n) = a(n+2) - 1
    sum(n) = if n < 0 then 0 else a(n) + b(n-1)    |
    result = sum(20)                               |
end
```

(here a and b are the w_i). The annotation already present is the induction hypothesis, to be used in deriving the desired annotation.

The transformations are straightforward.

One interesting feature of the rule is that the subproofs may involve any of the rules for deriving annotations; in particular, they may also use the induction rule, so that inductions may be nested.

8. REFERENCES

[1] Ashcroft, E.A. and Wadge, W.W. "Lucid, A Nonprocedural Language with Iteration", CACM 20, No. 7, pp. 519-526.

[2] _______________ "Structured Lucid", CS-79-21, Computer Science Department, University of Waterloo, June 1979.

[3] _______________ "Generality Considered Harmful - A Critique of Descriptive Semantics", CS-79-01, Computer Science Department, University of Waterloo, May 1979.

[4] Craig, W. "Logic in Algebraic Form", North Holland, 1974.

[5] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. "Initial Algebra Semantics and Continuous Algebras", JACM, 24, No. 1, pp. 68-95.

[6] Landin, P.J. "A Correspondence Between Algol 60 and Church's Lambda Notation", CACM 8, pp. 89-101, 158-165.

[7] Landin, P.J. "The Next 700 Programming Languages", CACM 9, No. 3, pp. 157-166.

[8] MacCarthy, J. "Towards A Mathematical Theory of Computation", Proceedings IFIP, 1962.

A LOGICAL PROGRAMMING LANGUAGE

Ed Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

June 1979, Revised March 1980

CS-79-20

A LOGICAL PROGRAMMING LANGUAGE

Ed Ashcroft
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

and

Bill Wadge
Department of Computer Science
University of Warwick
Coventry, England

Abstract

In this paper we consider a family of languages (USWIM) which is based on Landin's ISWIM (the individual languages being determined by appropriate continuous algebras of data objects and operations on these objects). We give a simple mathematical semantics for USWIM, and also give a system of program manipulation rules and a system of inference rules for reasoning about USWIM programs, the latter using a system of "program annotation" which allows inner, local environments to be discussed. USWIM is the basis on which Structured Lucid is built.

The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack.

[*P.J. Landin, 1966*]

0. INTRODUCTION

It is apparent that the goals of language designers and logicians are quite similar: to develop systems for precisely specifying objects and properties. In both cases this means the study and development of the syntax and semantics of purely formal, as opposed to natural, languages.

It is also apparent that logicians have been eminently more successful. The logicians' languages, such as predicate calculus, are simple, elegant and, above all, semantically well defined. Programming languages, by contrast, are complex, clumsy and, above all, semantically very poorly defined. It is often said that they are "illogical".

Furthermore, the languages of logicians were developed in conjunction with rules of inference, so that reasoning about properties of objects could proceed by simple finite manipulations completely within the language itself. By constrast, formal reasoning about programs, to the extent that it is possible at all, has to be carried out in a separate formal system in which the manipulations are performed on comments on, or a translation of, the original program.

The obvious conclusion is that logic (and mathematics in general) could be usefully applied to the study and design of programming languages. Few would dispute this; but there have always been two points of view about the relationships between logic and computer science.

One point of view sees mathematics as playing primarily a passive role, being used to describe, to model and to classify. The other point of view sees mathematics as playing primarily an active role, being used not so much to describe existing objects as to plan new objects.

These two approaches, which we might call the *descriptive* and *prescriptive* approaches [3], are well illustrated by two important papers by Landin, "A Correspondence between Algol 60 and Church's λ Notation" and "The Next 700 Programming Languages" [7, 8].

In the first Landin defines a translation from Algol 60 into the λ-calculus and so uses logic to describe Algol. In the second he begins with λ-calculus and develops a simple non-procedural language (ISWIM[†]), with a naturally defined construct (the "where" phrase[††]) which introduces local variables in a way similar to the Algol block, but which is actually based on the λ-calculus. Landin's first paper represents the descriptive approach and the second represents the prescriptive approach.

It is clear that if we want to develop computer languages having the elegance of mathematical languages it is the prescriptive approach that we must adopt.

It has become almost accepted without question that computer languages can not hope to have the simplicity we desire. For example, Scott and Strachey [11] say that "computer oriented languages differ from their mathematical counterparts by virtue of their *dynamic* character. An expression does not generally possess *one* uniquely determined value ... but rather the value depends on the state of the system at the time of

† If you See What I Mean.

†† We are using the term "phrase" rather than Landin's original "clause" because we shall consistently use "phrase" to mean a compound expression or term, and "clause" to mean a compound formula or assertion.

initialization of evaluation ... . Therefore the "algebra" of equivalences of such expressions need not be as "beautiful" as the well-known mathematical examples. This does not mean that the semantics of such languages will be *less* mathematical, only an order more complex". We feel that this attitude is a result of trying to mathematically describe existing languages. By taking the prescriptive approach, and basing new languages on existing mathematical languages, such as mathematical logic, more positive results can be obtained, and equivalences of expressions *can* be "beautiful".

Our goal here is to follow Landin's lead in the second paper mentioned above and develop an uncompromisingly logical language which has "facilities" for functions and scope.

The language, USWIM†, is in fact a minor variant of ISWIM. Landin, however, gave no direct semantics (ISWIM is a syntactic variant of a subset of the λ-calculus), and neither did he give an inference system for the verification of ISWIM programs. (He did give a system, of sorts, for transforming programs, but it is not very useful.) Of course, these omissions are not Landin's fault, because at that time semantics and program verification were in their infancy. We intend to fill in the gaps in Landin's treatment, and the fact that the semantics and transformation and inference rules turn out to be simple, natural and elegant is a vindication of Landin's mathematical approach to language design.

USWIM forms the basis of further development of Lucid [1], as is considered in [2], but we feel that it is also an interesting language in its own right.

† U See What It Means.

1. ISWIM AND USWIM

ISWIM is based on the where phrase, which is an expression qualified by auxiliary definitions. For example

$$\begin{aligned} x^2 + y^2 \quad \texttt{where} \quad & x = a + b \\ & y = a - b \\ \texttt{end} & \end{aligned}$$

is a typical where phrase. Such a phrase is a term, i.e. has a value; in the above example, it is the same as that of the term $(a+b)^2 + (a-b)^2$. These constructs can of course be nested, i.e. expressions occurring anywhere in a where phrase may contain where phrases. The variables defined by the equations in the right arm of the where phrase are the locals: their definitions apply only to the right arm and to the left-hand side expression of the phrase. In addition, functions are defined with the formal parameter list on the left hand side, e.g.

$$\begin{aligned} f(x, y) = x^2 - p \bullet q \quad \texttt{where} \quad & p = a \bullet y + b \\ & q = a \bullet x - b \ . \\ \texttt{end} & \end{aligned}$$

The formal parameters may represent functions, so "higher-type" functions can be defined. ISWIM also has a "whererec" phrase in which circular (recursive) definitions are allowed.

The main difference between USWIM and ISWIM is that the former is based on a slightly different phrase, namely the valof phrase. A valof phrase consists of a set of definitions (like those in the right arm of a where phrase) called the body, enclosed by the 'brackets' valof (i.e. "value of") and end . One of the variables defined in the body must be result , whose purpose is to indicate the value of the phrase. Functions are defined as in ISWIM, except that the formal parameters may *not* represent functions; there are no definitions of "higher-type" functions.

The following are legal valof phrases:

```
valof                            valof
     $x = a + b$                          $p = m^2 + 1$
     $y = a - b$                          $q = n^2 - 1$
  result = $x^2 + y^2$                $f(c, d) = 1 +$ valof
end                                                    $p = c^2 - d^2$
                                                   result = $p^2 + 2 \cdot p \cdot q$
                                                 end
                                      result = $f(3, q)$
                                  end
```

The most important difference between valof phrases and where phrases is that in the former recursiveness is assumed, i.e. the valof phrase corresponds to the whererec phrase of ISWIM. This eliminates one of the most confusing features of ISWIM, namely the fact that the same variable can refer to different things on the opposite sides of the same equation. Apart from this, there is another reason for preferring valof phrases to where phrases. USWIM is the basis of Structured Lucid [2], and, while it would be possible to base a Structured Lucid on the where phrases of ISWIM, it would then be necessary, as pointed out in [2], to consider the terms E and E **where end** as having different values (the second is a where phrase with an empty body).

2. SYNTAX OF USWIM

Landin's goal in designing ISWIM was to achieve a clear separation between two aspects of a programming language, the set of given or primitive things and the ways of expressing things in terms of other things. ISWIM is therefore not a single language but a family of languages, each of which is, in Landin's words, "a point chosen from well-mapped space". The coordinate of a point in this well-mapped space is the set of chosen primitives. Since ISWIM and USWIM are based on expressions, to specify the primitives we have to supply a domain of data objects, a collection of operations on these objects, and a collection of symbols used to denote the operations. In other words, a member of the USWIM family is determined by an algebra A ; we will call the corresponding language USWIM(A) . The syntax of USWIM(A) is determined only by the signature of A .

Suppose therefore that we are given an algebra A with signature Σ . That is, Σ is a collection of constant symbols of various arities ("individual constants" being of arity 0). We follow the usual terminology of symbolic logic and refer to the elements of Σ as "constant symbols", even though it is only those of arity 0 which are what computer scientists usually refer to as "constants". This is understandable when we realise that, for example, + , like 3, has the same meaning in all contexts.

We also assume that we have available an unlimited number of variables of various arities (these are what are usually called identifiers). The set of variables is the same for all algebras A .

The nullary variables will also be called "individual variables". Strictly speaking, the constant symbols and variables should be typed to indicate the number of arguments expected, but in practice we will omit these types. Non-nullary constant symbols will often be called operation symbols, and non-nullary variables will often be called function variables.

A USWIM program is simply a term, but to define the class of terms we must also define the classes of definitions and phrases simultaneously and inductively.

A *term* is either

(i) an n-ary constant symbol together with a sequence of n terms as operands (n will be zero if the symbol is an individual constant);

(ii) an n-ary variable together with a sequence of n terms as actual parameters (n will be zero if the variable is an individual variable);

or

(iii) a phrase.

A *phrase* consists of an unordered set of definitions, no two of which have the same definiendum, and exactly one of which has has the individual variable result as its definiendum.

A *definition* consists of a definiens which is a term, and of a term consisting of an n-ary variable f (the definiendum) together with an ordered set of n distinct individual (nullary) variables (the formal parameters). (f is called a *local* variable of the phrase.)

All variables which are not local variables of a phrase are called *global* variables of the phrase.

This, of course, is an abstract syntax of USWIM (in the sense of McCarthy [9]), analogous to Landin's abstract syntax for ISWIM. In our examples, (like those already given) we will use a fairly obvious *concrete* linear (or, more realistically, two dimensional) representation in which terms are written using infix notation, definitions are written as equations (with the definiens on the right), and phrases are written as sequences of equations enclosed by the keywords **valof** and **end.**† We will not give a precise definition of the concrete syntax. Such a definition would clearly not be particularly complex, but it can not just take the form of a context-free grammar because of the restriction that the formal parameters in a function definition be distinct and the restriction that no variable have two definitions in the same phrase.

In the metalanguage (i.e. when talking about terms in general) we will use expressions like $k(u_0, u_1, \ldots, u_{n-1})$ to denote the term consisting of the n-ary constant k together with the n terms $u_0, u_1, \ldots, u_{n-1}$ as operands. Similarly, we will use expressions like $\mathfrak{f}(u_0, u_1, \ldots, u_{n-1})$ to denote the term consisting of the n-ary variable $\mathfrak{f}$ together with the n terms $u_0, u_1, \ldots, u_{n-1}$ as actual parameters.

† The order of the definitions in a sequence will not be important.

3. SEMANTICS OF USWIM

The algebra A specifies not only the form of USWIM(A) but also its meaning, because A specifies a universe of data objects and assigns to each constant in Σ a meaning, which is an operation over the universe of A. Since USWIM allows arbitrary recursive definitions, we must, in assigning meaning to programs, be able to solve these definitions over the data domain. We therefore assume that the universe of A is a cpo and that the operations of A are continuous,† so that A is what the ADJ [5] school calls a "continuous algebra". The least element of A is $\perp$, which intuitively will be the "result" of non-terminating computations.

Suppose now that we have chosen a signature Σ and a Σ-algebra A.

Even though we have settled upon the meaning of the basic operations we still cannot in general conclude that such and such a term has such and such a value, because the term may have free variables (variables occurring in the term in positions at which there are no relevant definitions). For example, to know the value of the term

```
valof
          $d = a^2 + b^2$
          $e = a^2 - b^2$
   result = valof
                    result = $a \cdot q + b$
                         $a = 5 \cdot d$
                         $q = 7 \cdot e$
                end
end
```

† Thus, for example, the "equality operation" must be continuous, and will be denoted by eq to avoid confusion with the "equality relation" = used in definitions.

we must know the values of the variables a and b .

Thus the value of a term depends on both the values of the constant symbols (as represented by the algebra A) and the values of the variables. The latter we represent as an *environment*, by which we mean a function which assigns to each n-ary variable $\mathfrak{f}$ an n-ary function over the universe of A .

These environments are exactly what logicians call "assignments", a name which unfortunately already has other computer science connotations. This coincidence is unfortunate because the word "environment" itself is already widely used by computer scientists to refer to a wide variety of much more complicated structures.

There are two ways to express the fact that the meaning (value) of a term depends on the environment as well as on the algebra. One is to say that given an algebra and an environment, a term has such and such a value; and the second is to say that given an algebra, the value of a term is such-and-such a function from environments to values. In formal logic it is the first approach which is almost always used, whereas in the "Scott/Strachey" semantics it is almost always the second. On this question (as in most others) we choose the approach of the logicians over that of the computer scientists; in this particular instance, because it is notationally and conceptually simpler (avoids higher-type objects), and also because it avoids an unnecessary distinction between the algebra and the environment. Of course it is only the 'mathematical' nature of USWIM and the consequent simplicity of the notion of environment which allows us to make the choice.

For a given algebra A, the value of a term t in an environment E is defined inductively as follows:

a) If t is an n-ary constant k together with n arguments $u_0, u_1, \ldots, u_{n-1}$, the value of t in environment E is the result of applying the operation which A associates with k to the values of $u_0, u_1, \ldots, u_{n-1}$ in the environment E.

b) If t is an n-ary variable f together with n actual parameters $u_0, u_1, \ldots, u_{n-1}$, the value of t in the environment E is the result of applying the function associated with f (according to environment E) to the values of $u_0, u_1, \ldots, u_{n-1}$ in the environment E.

c) If t is a phrase the value of t in environment E is the value of **result** in the least† environment E' agreeing with E (except possibly for the locals of t) which satisfies the definitions in t.

An environment E satisfies a definition if, for all environments E' which differ from E at most in the values associated with the formal parameters of the definition, the value in E' of the left hand side term containing the definiendum is the same as the value in E' of the right hand side (definiens).

This, strictly speaking, is not a definition of the value of t in E since it assumes that least environments exist. However, if the operations in the algebra A are continuous, least environments *do* exist, so the above is a true statement about the value of t in E.

Since a USWIM program is just a term, the above is an informal

† The partial order on the data domain induces an order on the collection of environments.

description of the complete semantics of the language. It is a mathematical semantics rather than an operational one, that is, it does not specify the way one would actually compute the value of a program. If the data domain is reasonably simple (for example, the integers), it is in fact relatively straightforward to specify an operational semantics, using conventional Algol-like recursion-implementation techniques, that agrees with, i.e. realises, the mathematical semantics. In other data domains, especially those where the data objects are infinite, such as that of Lucid [1], the mathematical semantics corresponds to completely different operational concepts like iteration, data flow and coroutines.

4. SUBSTITUTION

The semantics of USWIM just given is simple and concise but cannot easily be applied directly in reasoning about programs. Instead, we use the semantics to justify purely syntactic manipulation rules and rules of inference. These rules, because they are purely syntactic (involve only text manipulation) can be used confidently without reference to any semantic or mathematical notions. The rules will be given here without proofs of their validity; such proofs exist, but it would be out of place to include them here, since they tend to be repetitive and lengthy.

In common with all formal logical systems, in order to specify rules of inference we must first define the concepts of free and bound variables, the operation of substitution, and the conditions under which substitution causes no clashes of scope.

The free occurrences of variables in a term are those which refer to objects external to the term itself. The value of a term depends only on the values of the objects referred to by these free variables. Substitution is the operation of replacing such free variables by terms. There are no clashes of scope caused by this operation provided the free variables of the terms being substituted still refer to external objects, i.e. are still free variables.

Informally, a bound occurrence of a variable in a term is an occurrence where the variable is a formal parameter of an enclosing definition or is a local variable of an enclosing phrase. All occurrences in the term which are not bound are said to be free.

More precisely, given any term t and any definition d, the *free occurrences* in t and d are defined as follows:

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then the free occurrences of variables in t are those corresponding to free occurrences in some u_i ;

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then the free occurrences in t are the initial occurrence of f together with occurrences corresponding to free occurrences in some u_i;

(iii) if t is of the form

```
valof
    e0
    e1
    .
    .
    .
    en
end
```

(where the e_i's are definitions)

then the free occurrences are those corresponding to free occurrences, in some e_i , of variables which are not locals of t ;

(iv) if d is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = a$$

then the free occurrences in d are the initial occurrence of f together with those free occurrences in a of variables other than the formal parameters $x_0, x_1, \ldots, x_{n-1}$ of d .

Any occurrence of a variable in a term or definition which is not a free occurrence is a *bound occurrence*. Any variable which occurs free in a term or definition is a *free variable* of the term or definition.

It is very common to want to talk about the result of replacing all free occurrences of a variable in a term by some other term. Since our variables are function variables, to do this replacement the (substituted) arguments of the variable being replaced must in turn be substituted into the term doing the replacing.

We say that the pair, consisting of the variable $\mathfrak{f}$ to be replaced (and individual variables $x_0, x_1, \ldots, x_{n-1}$ representing its arguments) and the term t to replace it (which usually contains free occurrences of the argument variables), is called an *assignment* . The assignment will be denoted by $\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow t$. A *substitution* is a set of assignments in which the variables to be replaced are all distinct.

We specify the result $t[S]$ of applying the substitution S to the term t in the following informal manner. We work outwards from the innermost terms of t , and whenever we find a free occurrence of a variable $\mathfrak{f}$ to be replaced (say $\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$ occurs in S), if the arguments of this occurrence are $u_0, u_1, \ldots, u_{n-1}$ (after possible substitution) then $\mathfrak{f}$ together with $u_0, u_1, \ldots, u_{n-1}$ will be replaced by a after simultaneously substituting u_i for x_i , $0 \leq i \leq n-1$.

More precisely, given any term t , any definition d and any substitution S :

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then $t[S]$ is

$$k(u_0[S], u_1[S], \ldots, u_{n-1}[S]) \ ;$$

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then $t[S]$ is

$$f(u_0[S], u_1[S], \ldots, u_{n-1}[S])$$

unless S has an assignment of the form

$$f(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

in which case $t[S]$ is

$$a[\{x_0 \leftarrow u_0[S], x_1 \leftarrow u_1[S], \ldots, x_{n-1} \leftarrow u_{n-1}[S]\}] .$$

(iii) if t is of the form

```
valof
    $e_0$
    $e_1$
    :
    $e_n$
end
```

(where the e_i's are definitions)

then $t[S]$ is

```
valof
    $e_0[\tilde{S}]$
    $e_1[\tilde{S}]$
    :
    $e_n[\tilde{S}]$
end
```

where $\tilde{S}$ is the result of removing from S all assignments to the locals of t .

(iv) if definition d is of the form

$$g(y_0, y_1, \dots, y_{n-1}) = b$$

then $d[S]$ is

$$g(y_0, y_1, \dots, y_{n-1}) = b[\hat{S}]$$

where $\hat{S}$ is the result of removing from S any assignments to the formal parameters $y_0, y_1, \dots, y_{n-1}$ of d .

Example

In the term

```
x + valof
          z = y + 3
       g(x) = if z > y then z else g(y + x)
     result = h(x + z)
    end
```

there are two free occurrences and two bound occurrences of x , three free occurrences of y , four bound occurrences of z , two bound occurrences of g , one free occurrence of h and one bound occurrence of result .

If we apply the substitution

$$\{x \leftarrow z,\ h(y) \leftarrow g(h(y + 1)),\ y \leftarrow (x + 2)\}$$

the result is

```
z + valof
          z = (x + 2) + 3
       g(x) = if z > x + 2 then z else g((x + 2) + x)
     result = g(h((z + z) + 1))
    end.
```

This substitution has several undesirable effects. After substitution, the two free occurrences of x become occurrences of z , but they refer to different objects, since the second is a local variable of the phrase. Similarly the three free occurrences of y become occurrences of "$x + 2$" but again they refer to different objects, since the x in the second and third is a formal parameter. These are examples of free occurrences of variables in the term doing the replacement becoming bound occurrences in the result of substitution. Another example of this is the substitution for h , where the free occurrence of the variable g becomes bound, since g is a local of the phrase.

We say that a term t *permits* a substitution S if none of these effects occur, i.e., if no free occurrence in a term being substituted becomes a bound occurrence in the result of the substitution.

In the same way, we say that a definition permits a substitution provided the definition permits the substitution and no free variables become formal parameters.

More precisely, for any term t , any definition d and any substitution S :

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then t permits S iff each u_i permits S ;

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then t permits S iff each u_i permits S , unless S contains an assignment of the form

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

in which case it is also required that a permit the substitution

$$\{x_0 \leftarrow u_0[S],\ x_1 \leftarrow u_1[S],\ \ldots,\ x_{n-1} \leftarrow u_{n-1}[S]\}$$

(iii) the term

```
valof
   $e_0$
   $e_1$
   .
   .
   .
   $e_n$
end
```

permits S provided each definition e_i permits the substitution $\tilde{S}$ described earlier, and provided that $\tilde{S}$ has no assignment of the form

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) \leftarrow a$$

with $\mathfrak{f}$ occurring free in some e_i and some local of t occurring free in a .

(iv) the definition

$$g(y_0, y_1, \ldots, y_{n-1}) = b$$

permits S provided b permits the substitution $\hat{S}$ described earlier and $\hat{S}$ does not contain an assignment of the form

$$\mathfrak{f}(x_0, x_1, \ldots, x_{m-1}) \leftarrow a$$

such that $\mathfrak{f}$ occurs free in b and a has a free occurrence of one of the formal parameters $y_0, y_1, \ldots, y_{n-1}$ of d that is not one of the arguments $x_0, x_1, \ldots, x_{m-1}$ of the assignment.

5. PROGRAM MANIPULATION RULES

With these technical matters taken care of we can proceed to consider the program manipulation rules.

The rules we present here are all transformation rules; they describe changes which can be made to a term without altering its meaning. If the term has free variables, the value is preserved no matter what value is assigned to those variables. This means that the rules can be used in any context, i.e. can be applied to subterms. As a result, our reasoning about programs can proceed as follows: we begin with the program P_0 and then define a sequence $P_1, P_2, P_3, \ldots$ of derived programs, each P_{i+1} the result of applying a rule to a subterm of P_i, until we arrive at a program P_n which more clearly has some desired property. Since the transformations preserve meaning, P_0 must also have the property. This stepwise or "incremental" approach to reasoning about programs closely resembles the approach mathematicians use in trying to prove something about, say, a system of differential equations. Mathematicians do not simply derive a series of assertions about the system; they transform the system itself, by adding new variables, eliminating old ones, integrating parts into closed forms, expanding power series, and so on. In logic itself most of the proof systems (e.g. the natural deduction systems) proceed by taking assertions apart and putting them together, but there is one system (Craig's Linear Reasoning [4]) which is based on a transformational approach. In Craig's system one proves an implication by transforming the hypothesis into the conclusion.

Another important characteristic of proofs in our system is that they are modular or 'nested' in that they follow the scope structure of the term to which they are applied. By this we mean that each application of a rule involves only the body of a single phrase. (To apply such rules we often have to use a rule for bringing definitions into a phrase.) This means that one can concentrate one's attention on one part of the program at a time, transforming its equations without altering, or even taking into account, definitions in inner or enclosing phrases.

The Rules

a) The first of our rules, the *import rule* , captures the idea that the definition of a variable applies to all free occurrences of the variable inside inner phrases, i.e. the idea that scope propagates inward. It says that we can add to any phrase a definition occurring (at the top level) in the smallest enclosing phrase provided the free variables of the definition are locals of the inner phrase, and are not formal parameters of the definition in whose right hand side the inner phrase occurs. For example, the rule allows us to transform

```
valof                                        valof

    f(x) = x^2 - y^2                             f(x) = x^2 - y^2
                                into
      y = p + 3                                    y = p + 3

  result = f(3 + y) - f(p)                   result = ((3 + y)^2 - y^2) - f(p)

end                                          end
```

Because of the way in which substitution works, this rule also captures the idea that the definitions with formal parameters are definitions of functions.

c) The *local renaming rule* expresses the idea that the local variables of a phrase are just dummy variables, that the actual variables chosen are not important, and that the enclosing phrases are 'shielded' from the definitions in the phrase. It says that in a phrase whose locals (other than result) are $v_0, v_1, v_2, \ldots, v_{m-1}$ we can choose any other sequence $w_0, w_1, \ldots, w_{m-1}$ of distinct variables (of corresponding arities) and replace all free occurrences of $v_0, v_1, \ldots, v_{m-1}$ in the definitions of the phrase by $w_0, w_1, \ldots, w_{m-1}$, provided the substitutions are permitted(i.e. as long as no v_i occurs where the corresponding w_i would be bound) and provided no w_i is a free variable of the original phrase. For example the rule allows us to transform

```
valof

        a = 3 - q(1, 1, 5)

      f(x) = x + valof

                        g(x) = x^2 - a

                          c = x^2

                      result = g(f(x)) + c

                 end

   q(r, s, t) = r^2 - a·r·s + f(t)

     result = a

end
```

```
valof                                        valof
        a = p + q                                    a = p + q
        b = p - q                                    b = p - q
   result = valof                               result = valof
                  b = a + 1       into                        a = p + q
             result = b^2                                     b = a + 1
           end                                           result = b^2
end                                                     end
                                             end
```

b) The *calling rule* expresses the idea that the definitions in a phrase really are equations, i.e. that the right hand side is really equal to the left hand side. It says that if a phrase contains the definition

$$f(x_0, x_1, \ldots, x_{n-1}) = a$$

then any definition of the form

$$d[\{h(x_0,x_1,\ldots,x_{n-1}) \leftarrow f(x_0,x_1,\ldots,x_{n-1})\}]$$

(where the substitution is permitted) can be replaced by

$$d[\{h(x_0,x_1,\ldots,x_{n-1}) \leftarrow a\}]$$

provided this latter substitution is permitted, that is, the free variables of a that are not formal parameters of the first definition remain free variables of the result of substituting in d. (This notational device, using the dummy variable h , allows us to "call" some occurrences of f and not others.) For example, the rule allows us to transform into

```
valof
            $b = 3 - k(1, 1, 5)$
        $v(x) = x + $ valof
                        $g(x) = x^2 - b$
                           $c = x^2$
                      result $= g(v(x)) + c$
                  end
    $k(r, s, t) = r^2 - b \cdot r \cdot s + v(t)$
        result $= b$
end
```

but we could not change f to g because the occurrence of f in the inner phrase is in a context in which g is bound. Also we can not rename the locals of the inner phrase to change c to a because the existing free occurrence of a in the inner phrase would become bound.

d) The *formal parameter renaming rule* is a similar renaming rule for formal parameters, which expresses the fact that they too are dummy variables.

e) The *amalgamation rule* is so-called because it allows us to amalgamate a term involving several phrases into a single phrase in which result is defined to be an analogous term, for example to transform the sum of two phrases into a phrase in which result is defined as a sum.

Suppose then that e is any expression, that $\nu_0, \nu_1, \ldots, \nu_{n-1}$ is a sequence of distinct individual variables

$p_0, p_1, \ldots, p_{n-1}$ is a sequence of phrases having (other than **result**) no locals in common, and none of whose locals is a v_i or occurs free in e . We define the phrase q as that formed by taking the definition **result** = e together with, for each i , all definitions in p_i with **result** replaced by v_i . Then the term

$$e[\{v_i \leftarrow p_i \mid i < n\}]$$

(the term involving the phrases) may be replaced in any context by q provided all the substitutions indicated are permitted.

For example, suppose that the expression e is $a - b \cdot z$, that n is 2 and v_0 and v_1 are a and b respectively and that p_0 and p_1 are

```
valof                              valof
          x = m + n                          r = m + k
          y = m•n          and               s = m - k
      result = x^2 - y                  result = r•s + 5
end                                end
```

Then the expression

$$\left(\begin{array}{l}\textbf{valof}\\ \qquad\qquad x = m + n\\ \qquad\qquad y = m \cdot n\\ \qquad \textbf{result} = x^2 - y^2\\ \textbf{end}\end{array}\right) - \left(\begin{array}{l}\textbf{valof}\\ \qquad\qquad r = m + k\\ \qquad\qquad s = m - k\\ \qquad \textbf{result} = r \cdot s + 5\\ \textbf{end}\end{array}\right) \cdot z$$

can always be replaced by

```
valof
            $x = m + n$
            $y = m \cdot n$
            $a = x^2 - y$
            $r = m + k$
            $s = m - k$
            $b = r \cdot s + 5$
      result = $a - b \cdot z$
end.
```

An interesting special case of the inverse† of this rule, which we shall call the ***result rule***, is obtained when n = 1 or n = 0. This tells us that we can replace any phrase of the form

```
valof
      ⋮
      result = $e$
      ⋮
end
```

by e itself (the conditions of the general rule require that no local of the phrase occur free in e).

f) We mentioned that mathematicians, in the course of manipulating a set of differential equations, might introduce new variables and

---

† In fact all the rules can be applied in either direction. This means that from a term t we can obtain a term s by the "inverse of rule r" provided that rule r could be applied to term s to yield term t .

discard old ones. There is exactly such a rule for USWIM. The *addition rule* allows us to add to a phrase *any* definition whatsoever provided only that the variable being defined does not occur free in the phrase, and is not already a local variable.

The inverse of the addition rule, called the *deletion rule*, allows us to eliminate a definition in the same circumstances. For example, the addition rule allows us to transform

```
valof                                    valof
    f(x, y) = x² + a•x•y + y²                h(x, y) = x² + y² - f(x, y)
        b = 3 - a                 into       f(x, y) = x² + a•x•y + y²
    result = f(2, b)                             b = 3 - a
end                                          result = f(2, b)
                                         end
```

and the deletion rule allow us to transform

```
valof                                    valof
        a = 3                                    b = 4
                              into
        b = 4                               result = x² - b•x
    result = x² - b•x                    end
end
```

g) Finally, there is a rule, the *basis rule*, which reflects the fact that USWIM is, in a sense, an extension of an algebraic language. It says that if the equation

$$t_1 = t_2$$

is true in the algebra A (here t_1 and t_2 are simple terms without phrases and the equation is considered to hold for all values of the free variables) then in any term *any* occurrence of t_1 may be replaced by t_2 . This allows us to carry over the laws of the data domain so that, for example,

```
valof                                          valof
    f(a, b) = (a + b)^2                            f(a, b) = a^2 + 2•a•b + b^2
                              can become
     result = f(3, 5)                               result = f(3, 5)
end                                            end
```

if the algebra A consists of the integers with the usual operations, in which the equation

$$(a + b)^2 = a^2 + 2 \cdot a \cdot b + b^2$$

holds for all a and b .

6. BINDING AND CALLING

If the language and rules of inference of the underlying algebra A are fairly standard, for example dealing with numerical quantities and interpreting the operation symbols in the usual way, then the rules of inference of USWIM just given are sufficient to successively transform any program (without free variables), whose value is defined, into a term without phrases denoting that value. This essentially corresponds to 'executing' the program.

For example, consider the program

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = f(3)
end.
```

Applying the calling rule we get

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = if 3 ≤ 0 then 1 else 3•f(3 - 1)
end.
```

Now applying the basis rule for properties of arithmetic we get

```
valof
        f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = 3•f(2)
end.
```

By repeating this process three more times we will get

```
valof
      f(n) = if n ≤ 0 then 1 else n•f(n - 1)
    result = 6
end.
```

Now applying the result rule, this term becomes simply 6.

The rules can also be used to answer questions about function-calling that in conventional programming languages are thought of operationally and are settled by conscious design decisions. These are the questions of the type of "parameter passing mechanism" used and whether the "binding" of global variables of functions is static or dynamic.

It is easy to see that the "parameter passing mechanism" is *not* call-by-value by considering the following program

```
valof
   f(x, y) = if x ≤ 0 then 0 else f(x-1, f(x, y))
    result = f(1, 0)
end.
```

If the calling "mechanism" were call-by-value, the value of this program would be $\perp$, since evaluation of $f(1, 0)$ in turn requires evaluation of $f(1, 0)$. However, it is easy to see that the rules allow us to generate the following sequence of equivalent programs:

```
valof
    f(x, y) = if x ≤ 0 then 0 else f(x-1, f(x, y))
     result = f(0, f(1, 0))
end
```

(by the calling and basis rules)

```
valof
    f(x, y) = if x ≤ 0 then 0 else f(x-1, f(x, y))
     result = if 0 ≤ 0 then 0 else f(0-1, f(0, f(1, 0)))
end
```

(by the calling rule)

```
0
```

(by the basis rule and the result rule).

Thus, the value of the program is 0 , not $\perp$.

As for the "binding" used, we can consider the following program:

```
valof
          a = 1
       f(x) = x + a
     result = valof
                         a = 2
                    result = f(2)
               end
end.
```

If the binding were dynamic, we would expect the value of this program to be 4, since it would be the inner definition of a that would be used by the "call" of $f(2)$. There are two ways to see that this is not the case. First, by the local renaming rule, we can rename the variable a either in the outer phrase or in the inner phrase, and in both cases the global variable of f will be the local defined in the outer phrase. Second, we can actually "execute" the program. This involves applying the calling rule to $f(2)$. However, the outer definition of result (which is the one, containing $f(2)$, which is in the phrase containing the definition of f) does not permit the substitution $f(x) \leftarrow (x + a)$ because the variable a , which is free in "$x + a$" , becomes bound in the resulting definition. Therefore we must first apply the local renaming rule, either to the inner or outer phrase, so that the substitution (which will have been changed if we renamed a in the outer phrase) is then permitted. It is then straightforward to see that the program reduces to 3.

Thus USWIM uses "static binding", as in fact all languages do which allow renaming of local variables.

7. INFERENCE RULES

The USWIM rules which we have described cannot by themselves be used to prove *assertions* about programs because these rules apply only to programs; they constitute a calculus of program transformations rather than a logic of programs. In this section we will present rules for proving assertions about programs. These rules are in no sense an axiomatic semantics of Lucid; they are justified by the formal semantics given earlier.

If we want to manipulate assertions about programs, we must extend our formal system by adding some class of assertions together with a collection of rules for deriving assertions. One way to do this is to take some existing logical system which already deals with terms and 'embed' USWIM in it by expanding the class of terms to include USWIM programs.

The most obvious choice for such a system is the first order language L whose individual variables are the nullary USWIM variables and whose operation symbols are taken from Σ. We generalize L to the language L^* by allowing a USWIM phrase whose free variables are all nullary to appear in a formula anywhere an ordinary term is permitted. In this 'extended' logic, assertions about programs are assertions about the values of programs and are just formulas in which the program appears; for example,

$$\forall i \left(\left(\begin{array}{l} \textsf{valof} \\ \qquad sq(n) = n \cdot n \\ \qquad \textsf{result} = sq(i-1) \cdot sq(i+1) \\ \textsf{end} \end{array} \right) = i^4 - 2 \cdot i^2 + 1 \right) .$$

There is no difficult in using the mathematical semantics of USWIM given earlier to extend the semantics of L , i.e. to define what it means for an L^*-formula to be true in a given environment.

Program-equivalence assertions are just equations between programs with their (common) input (free) variables universally quantified; for example,

$$\forall n \forall r$$

$$\left(\begin{array}{l} \textsf{valof} \\ \quad fac(x) = \textsf{if } x < 1 \textsf{ then } 1 \\ \qquad\qquad\quad \textsf{else } x \cdot fac(x-1) \\ \quad \textsf{result} = fac(n)/(fac(r) \cdot fac(n-r)) \\ \textsf{end} \end{array} \right) = \left(\begin{array}{l} \textsf{valof} \\ \quad h(x, y) = \textsf{if } x > y \textsf{ then } 1 \\ \qquad\qquad\quad \textsf{else } x \cdot h(x+1, y) \\ \quad \textsf{result} = h(n-r+1, n)/h(1, r) \\ \textsf{end} \end{array} \right)$$

There is, however, a serious problem with this approach, and that is that while the operation symbols of A can be immediately incorporated into L^* , the relation symbols cannot, they are just operations in A which happen to yield what A's interpretation of **if...then...else** regards as truth values. In fact, since Σ is a one-sorted signature it does not even make sense to talk about "relation symbols" in Σ .

We can, of course, define a many-sorted version of USWIM whose parameter would be a many-sorted algebra, and we could require that bool be one of these sorts and that A interpret it in some standard way. Nevertheless we could still not use operations in Σ of (result) type bool as relations in L^* because the value of such operations could be the 'undefined' element $\perp$ of A. This is possible because A is required to be a continuous algebra - and we cannot drop this requirement because our definition of the meaning of a USWIM phrase requires the existence of fixed-points of arbitrary recursive definitions.

There are two solutions to this problem. One is to distinguish completely between tests used in a program and actual relations over the universe of A.(this involves using two different forms for every basic test in A, e.g. distinguishing between $<$ and less-than , $>$ and greater-than , $\geq$ and greater-than-or-equal in the same way as between $=$ and eq). An alternate solution is to adopt a three-valued logical system plus additional rules specifying circumstances under which conventional two-valued reasoning may be used. The first method is formally simple but in practice is very cumbersome, whereas the latter requires a more elaborate formal specification but is much easier to use in practice. Here we shall use the latter method.

Because the use of full predicate logic involves complications which we would rather avoid here, we will instead use a simpler logical system. (Unavoidably this will not be as expressive and powerful as predicate logic.) We will use as the basis of our assertion language the equational algebraic system of signature Σ. We extend it by allowing, as generalized assertions, universally quantified equations between USWIM terms. More precisely, our assertions are of the form

$$\forall x_0 \; \forall x_1 \; \ldots \; \forall x_{n-1}(t_1 = t_2)$$

for some natural number n, some sequence $x_0, x_1, \ldots, x_{n-1}$ of (for simplicity) individual variables, and some terms t_1 and t_2. The above assertion e is true in E (in symbols $\models_E e$) iff the value of t_1 in E' is the value of t_2 in E' for any environment E' differing at most from E in the values E' gives to some of the x_i. The free variables of e are those variables other than the x_i which are free in t_1 or t_2. The assertion e permits a substitution S iff both t_1 and t_2 permit the substitution $\hat{S}$ obtained by removing from S all assignments to any x_i, provided no x_i occurs free in any assignment in $\hat{S}$, and provided both t_1 and t_2 permit $\hat{S}$. When the substitution is permitted the result $e[S]$ is

$$\forall x_0 \; \forall x_1 \; \ldots \; \forall x_{n-1}(t_1[\hat{S}] = t_2[\hat{S}]) \quad .$$

In future we shall refer to such universally quantified equations simply as "equations", and equations without quantifiers will be "basic equations".

In proving equations from other equations we can use the equational rules of instantiation* and replacement. More precisely,

(i) *Instantiation:*

from any equation of the form

$$\forall x_0 \; \forall x_1 \; \ldots \; \forall x_{k-1} d \quad ,$$

with d any basic equation, infer the equation

---

* In equational theories, instantiation is usually called substitution, but we have already used this term.

$$\forall y_0\ \forall y_1\ \ldots\ \forall y_{m-1} d[S]$$

for any permitted substitution S of the form

$$\{x_0 \leftarrow u_0,\ x_1 \leftarrow u_1, \ldots,\ x_{k-1} \leftarrow u_{k-1}\}$$

and $y_0, y_1, \ldots, y_{m-1}$ any variables not free in the original equation;

(ii) *Replacement*:

from an equation of the form

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}(t_0[v \leftarrow a] = t_1[v \leftarrow a])$$

and an equation

$$\forall y_0\ \forall y_1\ \ldots\ \forall y_{m-1}(a = b)$$

infer the equation

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}(t_0[v \leftarrow b] = t_1[v \leftarrow b])$$

provided the substitutions are permitted and any x_i appearing free in a or b is among the y_i .

Since we will be using other rules of inference as well, we need an extra *generalization rule* which allows us to infer

$$\forall y_0\ \forall y_1\ \ldots\ \forall y_{m-1} d$$

from d whenever no y_i occurred free in the assumptions from which d was derived. (Here d need not be a basic equation.)

These are exactly the ordinary equational rules except that (i) the terms may contain USWIM phrases, and (ii) quantification is explicit. In ordinary equational algebra all variables are implicitly universally quantified, but we have made quantification explicit in order to avoid a distinction between variables in programs and variables in assertions. We therefore need a third rule (which we will not formulate precisely) allowing us to permute the order in which the variables are quantified in an equation.

These rules are sound in the following sense: if we can, using these rules, derive an equation e from a set $d_0, d_1, \ldots, d_{n-1}$ of equations, then e is true in every E in which all the d_i are true; in symbols,

$$d_0, d_1, \ldots, d_{n-1} \models e \quad .$$

Our program transformation system allows an extra rule which express precisely the fact that transformed programs are equivalent. If our transformation rules allows us to transform the term t_1 into the term t_2 we may infer the equation

$$\forall x_0 \; \forall x_1 \; \ldots \; \forall x_{n-1}(t_1 = t_2)$$

for any individual variables $x_0, x_1, \ldots, x_{n-1}$.

The substitution and replacement rules are general because they are sound no matter what the algebra A is. In studying a particular instance of the USWIM family, however, it is possible to use particular but very useful rules of inference valid only with reference to the particular algebra. An important example is the mathematical induction rule of the algebra N of natural numbers (with $\Omega \in \Sigma$ and $N(\Omega) = \perp$). To prove an equation $\forall v \; e$ from a set of assumptions it is enough to prove the equations

$$\mathcal{e}[\{\nu \leftarrow \Omega\}] \qquad \text{and} \qquad \mathcal{e}[\{\nu \leftarrow 0\}]$$

from the assumptions, and also prove $\mathcal{e}[\{\nu \leftarrow \nu+1\}]$ from the set of assumptions with $\mathcal{e}$ added (as the induction hypothesis), provided ν is not free in any assumption in the set.

The problem that remains with the "embedding" approach just described is that assertions about programs are just equations in which the programs appear as terms, so that they are, in fact, assertions about the *values* of programs. Programs are therefore treated as black boxes whose input-output activity is all that can be discussed. Now it is certainly true that, in the end, the goal of our reasoning is a statement about the meaning of a whole program, i.e. about its value; but in the course of working towards that goal we might want to make assertions about parts of the program. For example in showing that

```
valof
        $d = a_1 \cdot b_2 - a_2 \cdot b_1$
        $e = c_1 \cdot b_2 - b_2 \cdot c_1$
    result = $e/d$
end
```

has a certain value we would naturally want to prove that (say) $d = k+1$ (and so is positive) inside the phrase. Since this is a statement about the phrase's internal environment, we cannot do this literally, with the 'embedding' system. In such a system,the formalisation of reasoning about the interior of a program involves taking the program apart and putting it back together again, just as described in Section 5.

Clearly, what is required is some system which allows modular, local reasoning *about* a program in the same way that the transformation rules given in Section 5 allow modular, local transformations *of* a program. In reasoning about programs or in documenting them programmers often "annotate" them with comments (written, say, to the right of the text) which refer to the 'local' values of program variables. The system we propose is simply a formalisation of this idea.

We first define two classes of syntactic objects, which we respectively call *annotated terms* and *annotated definitions* , as follows:

An *annotated term* is either

(i) a constant in Σ together with an appropriate number of operands, each of which is an annotated term;

(ii) a variable together with an appropriate number of actual parameters, each of which is an annotated term;

(iii) an annotated phrase, i.e. a set of annotated definitions, one of which has result as its definiendum and no two of which have the same definiendum *together with* a set of universally quantified equations. (This set of equations is called the *paraphrase* of the phrase, and its elements are called *annotations* .)

An *annotated definition* is similar to a definition (with the definiens and definiendum defined accordingly) except that the definiens is an annotated term.

We will extend our concrete two-dimensional representation of terms by writing annotations to the right of the phrase to which they are attached, with a vertical line between:

```
valof
        $b = 3 \cdot k$          |    $k = j+1$
        $c = 2 \cdot k^2$        |
    result = valof
                $d = b^2 - 4 \cdot c$               |   $b^2 = 9 \cdot k$
             result = $(-b + sqrt(d))/2$      |   $d = k^2$
             end
end
```

An annotated phrase has two 'values', its value as a term (a 'data' value), and its value as an assertion (a truth value). The data value is simply the value of the term which results when the annotations are 'thrown away',and the truth value is the conjunction of the annotations in the phrase, each considered as referring to the local environment of the phrase to which it is attached. The idea that annotations refer to local environments can be made precise as follows. Given any annotated term t , any annotated definition d and any environment E :

(i) if t is of the form

$$k(u_0, u_1, \ldots, u_{n-1})$$

then t is true in E iff each u_i is true in E ;

(ii) if t is of the form

$$f(u_0, u_1, \ldots, u_{n-1})$$

then t is true in E iff each u_i is true in E ;

(iii) if t is of the form

```
valof
    e_0     |   p_0
    e_1     |   p_1
     :      |    :
    e_n     |   p_{m-1}
end
```

let E' be the least environment which agrees with E (except possibly for the locals of t) which satisfies the definitions obtained by de-annotating the e_i ; then t is true in E iff each p_j is true in E' and each e_i is true in E' ;

(iv) if annotated definition d is of the form

$$f(x_0, x_1, \ldots, x_{n-1}) = a$$

then d is true in E iff a is true in every environment E' which agrees with E except possibly for the values given to the formal parameters $x_0, x_1, \ldots, x_{n-1}$ of d .

In future, we will drop the word "annotated", and when we want to indicate the part of a phrase, term etc. without its annotation, we will use the word "de-annotated".

Now, if programs are annotated, we cannot use the program manipulation rules of Section 5 without first modifying them to take account of annotations. We will not go through all these modifications here, because they are all rather obvious. The general consideration is that the free variables of a phrase now include the free variables of the paraphrase that are not locals of the phrase, and no changes to the de-annotated phrase can be allowed which change the meanings of the annotations. This usually means that the changes made to a de-annotated phrase must also be made to its paraphrase (as, for example, in the local renaming rule).

As well as the program manipulation rules, we now also have rules that deal explicitly with annotations†.

I. The *discard rule* allows us to remove any annotation from any paraphrase.

II. The *consequence rule* allows us to add to any paraphrase any equation which is an A-consequence of the annotations already in the paraphrase. In other words, if

$$d_0, d_1, \ldots, d_{n-1} \models e$$

and each d_i is in a particular paraphrase, then e may be added to the paraphrase. In particular, if e can be obtained from the d_i using general or particular rules of inference, it may be added to the paraphrase.

III. The *definition rule* allows us to add to a paraphrase the equation

$$\forall x_0 \forall x_1 \ldots \forall x_{n-1} (f(x_0, x_1, \ldots, x_{n-1}) = a)$$

whenever the phrase contains the definition

$$f(x_0, x_1, \ldots, x_{n-1}) = a \ .$$

(Alternatively, we could allow the calling rule to be applied to annotations as well as to de-annotated phrases.)

† These inference rules, unlike the program manipulation rules, are not invertible.

IV. The *import rule* allows us to add to the paraphrase of an inner phrase any equation in the paraphrase of the immediately enclosing phrase provided it has no free occurrences of variables local to the inner phrase, or of the formal parameters of the definition containing the inner phrase. (This rule is simply a version, for annotations, of the import rule for program manipulation, so using the same name should not cause any confusion.)

V. We can also generalise the amalgamation/result rule, so that we can replace q by $e[\{v_i \leftarrow p_i \mid i < n\}]$ (or vice versa) in annotations as well as de-annotated terms.

VI. The *export rule* says that if we have annotation e in a phrase p whose enclosing definition has formal parameters $x_0, x_1, \ldots, x_{n-1}$, and the free variables of e are not locals of the phrase (other than, possibly, result), then we can add to the paraphrase of the enclosing phrase the annotation

$$\forall x_0 \, \forall x_1 \, \ldots \, \forall x_{n-1} \, e[\{\mathtt{result} \leftarrow p\}] \, .$$

The rules are sound in this sense: if an annotated term t' can be obtained from an annotated term t , then t' is true in any environment in which t is true.

The rules just described for annotated terms all have the property that their use involves only a *one-way* flow of information *from* the de-annotated program *to* the annotations. If t' can be obtained from t using the rules of inference and the manipulation rules , the corresponding de-annotated terms will have the same value, but the de-annotated version of t' could have been obtained from the de-annotated version of t using just the program manipulation rules. The annotation rules described so far are useful for verification but not for program transformation. Without rules for information flow in the opposite direction, we are restricted, as we are by existing verification techniques, to being unable to modify programs, on the basis of properties already proved, in order to ease the proofs of subsequent properties.

VII. There *is* a rule which allows annotations to be used to change the de-annotated program. The *modification rule* says (in its simplest form) that if a paraphrase contains the basic equation $a = b$, then certain occurrences of term a in definiens in the phrase may be replaced by term b . This simplest form of the rule would not allow us to substitute b for a in contexts in which some of the free variables of a or b are formal parameters of a definition in the phrase (it clearly would be incorrect to do so, since the assertion $a = b$ can not be referring to the formal parameters). A more general form of the rule allows us to substitute b for certain occurrences of a in this situation if the paraphrase contains

$$\forall x_0, \forall x_1, \ldots, \forall x_{n-1}\ a = b$$

where $x_0, x_1, \ldots, x_{n-1}$ are the formal parameters of the definition in question.

The restrictions on the occurrences of a are not just those necessary to avoid a clash of variables; it is also necessary to take the dependencies of the various locals of the phrase into account, in order to avoid perturbing the least local environment of the phrase.

To understand the necessity of the restriction, consider a program in USWIM(N) containing the definition

$$f(n) = \texttt{if } n < 1 \texttt{ then } 1 \texttt{ else } n \cdot f(n-1)$$

we might deduce the true annotation

$$\forall n(\texttt{if } n < 1 \texttt{ then } 1 \texttt{ else } n \cdot f(n-1) = (n + (1-n)) \cdot f(n-1))$$

(recall that for example, $1 - 3 = 0$ in N), and if we could apply this to the program using the more general form of the modification rule, the definition of f would become

$$f(n) = (n + (1-n)) \cdot f(n-1) \quad .$$

The problem now is that the factorial function is no longer the *least* solution of this definition; the least solution is the function whose value is always $\bot$. Many and subtle examples of this phenomenon could be given.

We see then that we can change the meaning of the program by substituting equals for equals, because the new definition may have *less* information than the old one. One way to ensure the correctness of a substitution is to require that the equation to be applied is a substitution instance of a more general equation, no free variable of which depends (in the phrase) on the definiendum of the definition to which it is applied.

More precisely, suppose that p is an annotated phrase containing a definition of the form

$$\mathfrak{f}(x_0, x_1, \ldots, x_{n-1}) = a[\{v \leftarrow t_1\}]$$

(where the substitution is permissible) and that the equation

$$\forall x_0\ \forall x_1\ \ldots\ \forall x_{n-1}(t_1 = t_2)$$

can be obtained from an annotation e in the paraphrase of p by one application of the instantiation rule. Then the definition may be changed to

$$f(x_0, x_1, \ldots, x_{n-1}) = a[\{v \leftarrow t_2\}]$$

provided the substitution in permissible and there is no free variable w of e such that $\langle w, f\rangle$ is in the transitive closure of the relation

$$\{\langle y, z\rangle \mid y \text{ and } z \text{ are locals of } p \text{ and } z \text{ occurs free in the definition of } y\} .$$

The requirement essentially implies that the definition being changed could not usefully have been used to derive the equation being used.

This requirement prevents the erroneous modification mentioned earlier because f occurs free in the equation and $\langle f, f\rangle$ is in the transitive closure of the relation for the phrase. On the other hand, the equation

$$\forall n(n \cdot f(n-1) = f(n-1) \cdot n)$$

can be used to modify the program if the annotation

$$\forall x \forall n(n \cdot x = x \cdot n)$$

is first attached to the phrase.

We will illustrate our transformation and manipulation rules by showing that the following inefficient USWIM(N) program to compute the sum of the first n squares,

```
valof
    $sum(m)$ = valof
                   $sq(k)$ = if $k < 1$ then 0 else $sq(k-1) + (2 \cdot k-1)$
                       $b = sq(m) + sum(m-1)$
                  result = if $m < 1$ then 0 else $b$
               end
    result = $sum(n)$
end,
```

is equivalent to the term $n \cdot (n+1) \cdot (2 \cdot n+1) \div 6$.

The first step is to use the definition rule to add the equation

$$\forall k(sq(k) = \textbf{if } k < 1 \textbf{ then } 0 \textbf{ else } sq(k-1) + (2 \cdot k-1))$$

as an annotation to the inner phrase. Next we use mathematical induction to prove

$$\forall i(sq(i) = i^2)$$

by induction on i :

The 'base' steps, namely proving $sq(0) = 0^2$ and $sq(\Omega) = \Omega^2$, are straightforward, and the induction step involves proving

$$sq(i+1) = (i+1)^2$$

from the assumption $sq(i) = i^2$ (notice that i does not occur free in our other assumptions). To do this we instantiate the equation derived from the definition of sq , giving

$$sq(i+1) = \mathtt{if}\ (i+1) < 1\ \mathtt{then}\ 0\ \mathtt{else}\ sq((i+1) - 1) + (2\cdot(i+1) - 1) \quad .$$

Replacements using equations like

$$\forall i((i+1) - 1 = i)$$

give us

$$sq(i+1) = \mathtt{if}\ (i+1) < 1\ \mathtt{then}\ 0\ \mathtt{else}\ sq(i) + 2\cdot i+1 \ .$$

Replacing $sq(i)$ by i^2 (i.e., by using the induction hypothesis) we get

$$sq(i+1) = \mathtt{if}\ (i+1) < 1\ \mathtt{then}\ 0\ \mathtt{else}\ i^2+2\cdot i+1$$

and since

$$(i+1)^2 = \mathtt{if}\ (i+1) < 1\ \mathtt{then}\ 0\ \mathtt{else}\ i^2+2\cdot i+1$$

is true in N , we can replace and get

$$sq(i+1) = (i+1)^2$$

as required.

The induction rule therefore allows us to add $\forall i\ sq(i) = i^2$ to our annotations, and instantiation gives $sq(m) = m^2$. Since m is not a local of the phrase and sq is not defined in the phrase directly or indirectly in terms of b , we can use the modification rule to replace the occurrence of $sq(m)$ by m^2 in the definition of b . Then we use the calling rule on the occurrence of b in the definition of result giving

```
valof
    $sum(m)$ = valof
                    $sq(k)$ = if $k < 1$ then 0 else $sq(k-1) + (2 \cdot k-1)$
                        $b = m^2 + sum(m-1)$
                    result = if $m < 1$ then 0 else $m^2 + sum(m-1)$
              end
    result = $sum(n)$
end
```

(after discarding the annotations).

The next step is to use the result rule and replace the entire inner phrase by the definiens of result , yielding

```
valof
    $sum(m)$ = if $m < 0$ then 0 else $m^2 + sum(m-1)$
    result = $sum(n)$
end.
```

We can now use annotations much as we did before to prove the equation

$$\forall m(sum(m) = m \cdot (m+1) \cdot (2 \cdot m+1) \div 6)$$

by induction on m, then use substitution and modification to get

```
valof
    $sum(m)$ = if $m < 0$ then 0 else $m^2 + sum(m-1)$
    result = $n \cdot (n+1) \cdot (2 \cdot n+1) \div 6$
end.
```

Using the result rule finally gives the term $n \cdot (n+1) \cdot (2 \cdot n+1) \div 6$. Since we have transformed the original program p into this term, and since we did not use any assumptions about n, we conclude

$$\forall n(p = (n \cdot (n+1) \cdot (2 \cdot n+1) \div 6)$$

is true in N.

The USWIM rules do not in themselves make verifying programs any easier mathematically (although they make it easier notationally). The real significance of the rules is that they allow the proofs derived to be completely precise, i.e. broken down into a series of small steps each of which is the application of a simple rule. Naturally this degree of precision would be possible only with the aid of a mechanical proof checker capable of 'interpolating' simple steps. Such a checker/verifier based on USWIM would be no more complicated than many existing systems, and would allow a user to perform sophisticated manipulations with complete confidence.

8. REASONING BY FIXPOINT INDUCTION

We have mentioned that the definitions in a phrase can be thought of as true assertions about the locals, and two of the manipulation rules given can (as we pointed out) be considered as justifications of this way of thinking. Neither of these rules, however, make use of the fact that the environment inside a phrase is the *least* one which makes the equations true. The two rules given express the fact that the values of the locals are a fixed-point of the equations, but they do not express the fact that they are the *least* fixed point.

This is not as serious a deficiency as it might seem, because in very many cases the fixed-point is unique anyway. Sometimes, however, there is more than one fixed-point, and,even when there isn't, it is often easier to derive a particular result using minimality even when **minimality is not strictly necessary.**

We will describe a USWIM rule which is based on the "Fixpoint Induction" principle (see [10]) which has the useful property that it can be stated without explicit reference to the approximation relation in the domain in question.

The Fixpoint Induction rule is the following: given any cpo, any continuous function τ over the cpo with least fixed point κ , and any property p : to show $p(\kappa)$ show $p(\perp)$ and then show that $p(\alpha)$ implies $p(\tau(\alpha))$ for any α in the cpo. The rule is valid provided that p is "admissible" in the sense that the lub of a chain of elements in the cpo possesses the property whenever every element in the chain does.

The rule itself is a meta-rule, i.e. it refers to semantic objects. The corresponding USWIM rule is an object rule in that it refers to syntactic objects and gives conditions under which annotations can be added to terms. Suppose then that t is a phrase, that $\nu_0, \nu_1, \ldots, \nu_{n-1}$ are any of the locals of the phrase, and that q is some universally quantified equation about the ν_i and other variables which we wish to prove by induction on the ν_i . (q is bound to be admissible.) We assume that Σ has nullary symbol Ω which A interprets as $\perp$, and that $\omega_0, \omega_1, \ldots, \omega_{n-1}$ are variables which are not local to the phrase and do not occur free in any annotation in the phrase. Then in order to justify adding q to the phrase we must

(i) be able to add q to the paraphrase of the phrase t' formed by changing the definition of each ν_i to

$$\nu_i(x_0, x_1, \ldots, x_{r_i-1}) = \Omega$$

(where r_i is the arity of ν_i);

(ii) be able to add q to the paraphrase of the phrase t'' which results from applying in t the substitution

$$\{\nu_i(x_0, x_1, \ldots, x_{r_i}) \leftarrow \omega_i(x_0, x_1, \ldots, x_{r_i}) \mid i < n\}$$

to the definiens of the ν_i , and adding as an annotation the result of applying the above substitution to q .

As an example of the application of the rule, suppose that we wish to add the annotation

$$\forall n \; sum(n) = fib(n+2) - 1$$

to the phrase

```
Valof
    $fib(n)$ = if $n \nless 2$ then 1 else $fib(n-1) + fib(n-2)$
    $sum(n)$ = if $n < 0$ then 0 else $fib(n) + sum(n-1)$
    result = $sum(20)$
end
```

where the algebra in question consists of the integers (plus $\perp$) and the usual operations.

We must first show that we can add the given annotation to the phrase

```
valof
    $fib(n) = \Omega$
    $sum(n) = \Omega$
    result = $sum(20)$
end;
```

and then show that the same annotation can be added to the annotated phrase

```
valof
    $fib(n)$ = if $n < 2$ then 1 else $a(n-1) + a(n-2)$    | $\forall n \; b(n) = a(n+2) - 1$
    $sum(n)$ = if $n < 0$ then 0 else $a(n) + b(n-1)$      |
    result = $sum(20)$                                     |
end
```

(here a and b are the w_i). The annotation already present is the induction hypothesis, to be used in deriving the desired annotation.

The transformations are straightforward.

One interesting feature of the rule is that the subproofs may involve any of the rules for deriving annotations; in particular, they may also use the induction rule, so that inductions may be nested.

8. CONCLUSION

The language we have specified, and for which we have given semantics, rules of inference and manipulation rules, is similar to Landin's USWIM. It is based upon expressions (terms) and the semantics of these expressions is not "an order more complex" than for mathematical expressions. In fact the "algebra" of equivalences of these expressions is as "beautiful" as those of λ-calculus or predicate calculus, for example.

The semantics of USWIM is given in a way which separates the data aspects of the language from the computational aspects. This fits in well with current work in, for example, abstract data types [6]. Using well thought-out data structures, like arrays or the streams in Basic Lucid [1], gives languages of considerable practical power.

USWIM is a spartan language, in the sense that it has only one basic "mechanism", recursion. It is known that recursion is more powerful than iteration, for example, and it is appropriate to extend USWIM by adding constructs, and possibly modifying the semantics, in order to add less powerful "mechanisms", so that simple algorithms can

can be expressed in a simple way. An example of this can be seen in the paper "Structured Lucid" [2].

In summary, we feel that USWIM is a way of expressing communications to computers that possess Landin's "certain simplicity".

9. REFERENCES

[1] Ashcroft, E.A. and Wadge, W.W. "Lucid, A Nonprocedural Language with Iteration", CACM 20, No. 7, pp. 519-526.

[2] ________________ "Structured Lucid", CS-70-21, Computer Science Department, University of Waterloo, June 1979.

[3] ________________ "R_x for Semantics", CS-79-37, Computer Science Department, University of Waterloo, December 1979.

[4] Craig, W. "Linear Reasoning. A New Form of the Herbrand-Gentzen System", JSL 22, No. 3, pp. 250-268.

[5] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. "Initial Algebra Semantics and Continuous Algebras", JACM 24, No. 1, pp. 68-95.

[6] ________________ "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types", Current Trends in Programming Methodology & Data Structuring (R. Yeh, ed.), Prentice Hall, 1976.

[7] Landin, P.J. "A Correspondence Between Algol 60 and Church's Lambda Notation", CACM 8, pp. 89-101, 158-165.

[8] Landin, P.J. "The Next 700 Programming Languages", CACM 9, No. 3, pp. 157-166.

[9] McCarthy, J. "Towards a Mathematical Theory of Computation", Proceedings IFIP, 1962.

[10] Manna, Z. "Mathematical Theory of Computation", McGraw-Hill, 1974.

[11] Scott, D. and Strachey, C. "Towards a Mathematical Semantics for Computer Languages", Proc. Symp. Computers and Automata (J. Fox, ed.), Polytechnic Institute of Brooklyn Press, New York, 1976.