On Homomorphisms, Correctness, Termination, Unfoldments, and Equivalence of Flow Diagram Programs*

J. A. Goguen, Jr.

Computer Science Department, University of California, Los Angeles, California 90024

Received February 28, 1973

1. INTRODUCTION AND SUMMARY

This paper presents a definition of homomorphism for programs and applies it to giving simplified proofs of correctness, equivalence, termination, and other properties of programs. We mean "simplified" in the sense that no manipulations with first (or higher) order logic, or other formal systems, are required, and that human beings sometimes seem to be able to produce shorter and more systematic (rigorous) proofs than by many other methods. We use only the most elementary set theory for verifying programs, although some fancier algebra is needed for the proofs of validity of the verification methods. Moreover, we have included a number of examples to illustrate the power of the methods, and have defined all the less familiar algebraic concepts. It should be noted that this general line of research was initiated by Burstall [1].

The general idea of a homomorphism \( h : P_0 \rightarrow P_1 \) of programs is that it represents a combination of the following two special cases of relationship between programs: (1) \( P_0 \) is "simulated" by \( P_1 \), meaning everything \( P_0 \) does can also be done by \( P_1 \), though possibly in a different way; and (2) \( P_1 \) is a "cruder version" of \( P_0 \), meaning that the results of \( P_1 \) can be systematically obtained from those of \( P_0 \). Roughly speaking, in case (1) the homomorphism \( h : P_0 \rightarrow P_1 \) is an inclusion, while in case (2) it is a quotient or "projection." Homomorphisms of type (1) are useful in proving correctness by considering a "generic input," while those of type (2) can be useful in proving termination by eliminating irrelevant information from the computation.

In each case, the crucial point for the usual applications is that the existence of the homomorphism, though seeming to involve infinitely many conditions can actually be verified with a finite number of relatively simple conditions. This uses the freeness of a certain path or flow construction. However, the general methods apply just as well to infinite programs, though verification of an infinite number of conditions may be

* This research was supported in part by National Science Foundation Grant No. GJ33007X.
needed. In reasonable cases, the program is sufficiently regular to permit verification via finitely describable "condition schemes," which in fact turn out to be the so-called "regular expressions" of standard automaton theory.

One such case arises from the process of "unfolding" a graph into its "best possible" loop-free form. The unfoldment is in general an infinite graph, but is finitely describable if the original graph is finite. We show the unfoldment is "best possible" in the sense of satisfying a certain "universal property," and we also show that the unfoldment of a flow program is "flow equivalent" to the original program. This provides an interesting technique for proving equivalence of programs having quite different loop structures. We illustrate this technique with a not entirely trivial equivalence problem having some previous history in the literature, and suggest an approach to the partial automation of equivalence proofs. We also argue that equivalence preserving transformations provide an appropriate general approach to program semantics for many applications.

It is a commonplace in modern mathematics (see [13]) that one should seek out and maximally utilize the structure-preserving homomorphisms of any class of objects under study. This has not been so easy in computer science, presumably because the subject is so new and its strongest intuitions so practical (see [8] for further discussion, and many further examples). In the present case, the notion of program homomorphism is a special case of an earlier notion of homomorphism for "general systems" (see [5]). This fact reinforces the intuition as to the correctness of the definition, and also permits the application of a number of previously proved results, such as those on multilevel hierarchical organization [6]. However, the present paper is entirely self-contained, and the various applications of the general theory of systems to program semantics are deferred to later papers. These include hierarchically structured programs and program schemes, as mentioned in the preliminary version [7] of this paper. To have included them here would have required much more in the way of mathematical apparatus than it seemed desirable to introduce all at once, as well as much more space.

We use algebraic methods throughout, particularly the notions of category, functor, and natural transformation, which are fully defined in the text. Further topics and many computer science examples can be found in [8]; and many standard mathematics examples are in Mac Lane's definitive text [13]. We do not assume familiarity with Burstall [1], but the reader may find comparison fruitful; in particular, [1] contains versions of a number of the results appearing here before Sections 7 and 8, though without the unifying concept of homomorphism. In addition to generalizing [1] to sets of entry and exit nodes, we have provided simpler and more rigorous proofs, and a number of new results, particularly on unfoldments, termination and equivalence. We discuss both deterministic and nondeterministic programs.

It is conceivable that a suitable interactive computer implementation of a proof checker based on the methods described here would enable a skilled programmer to routinely verify the correctness of fairly large programs. However, there are some
difficulties inherent in this proposal. For one, a suitable computer language with set theoretic capabilities would be needed; but J. Schwartz [16] has developed one such language, SETL, which might be used as a basis. Another difficulty is that complicated graph manipulation routines would be needed, especially for the infinite, though finitely describable, graphs which arise from unfolding recursions or loops. Two major difficulties of totally automatic program verification are overcome by operation in an on-line interactive mode. These difficulties are: (1) the notorious weakness of theorem provers, especially in situations which are not highly specific; and (2) the undecidability of the verification problem (i.e., there is no algorithm guaranteed to work uniformly for all programs). We envision even a fairly weak theorem prover as being of some help in the interactive situation, since only the verification of small steps would be asked of it. Moreover, it is possible that the inherent hierarchical structure of our situation, when suitably formalized, will help the theorem prover with its task.

A third difficulty is only partially overcome. Some of the most common bugs in programs arise from sources which are difficult to formalize, such as overflows and vague specifications. The difficulties of formalization are that different computer-operating system-compiler configurations may actually do somewhat different things, that these things may be very complex, and that just what they are is not usually well documented or easily discovered (partly because the system itself may not have been proved correct). The complexity difficulty is somewhat ameliorated by a computer's ability to automatically handle large expressions. The fact that one of the goals behind program verification work is better documentation and reliability of computer system implementations encourages us to think that partial success with programs written in (say) ALGOL might eventually be transferred to yield better ALGOL compilers.

More technically now (and using terms to be defined quite precisely later in the paper), the “shape” of a program is a graph $G$, and a program is a functor $\text{Pa}(G) \rightarrow \text{Pfn}$ assigning partial functions to the paths or flows in $G$, as in [1]. Our homomorphisms of programs do not require them to have exactly the same shape, and this permits a more satisfactory and general notion of simulation than that of Burstall [1] or Milner [14]; on the other hand, the shapes allowed are not without restriction, as flows of control must be preserved by flow diagram homomorphisms. Our discussion of correctness uses simulation and generally follows Burstall’s while clarifying and generalizing it in several aspects; it therefore also justifies, generalizes and clarifies Floyd’s method [3]. Our discussions of termination and equivalence use various somewhat different special kinds of homomorphisms, including projections. These also appear in the relationship between a program and its loop-free unfoldment, which can be used in proving equivalence. The paper gives a number of examples illustrating these points.

We wish to thank Barry Rosen, Susanna Ginali and Jack Schwartz for their useful suggestions and comments on the material of this paper, and Rod Burstall for providing the stimulus which started it.
2. THE PATH CATEGORY

We begin with freeness of the path construction, which is basic to all further work in this paper. Let $G$ be a (directed) graph, i.e., a quadruple $\langle G, E, \partial_0, \partial_1 \rangle$, where $\partial_i : E \to |G|$ are the source ($i = 0$) and target ($i = 1$) functions from the edges to the nodes (or vertices) of $G$. As a notational convention, we often let $\bar{G}$ denote the set $E$ of edges of $G$, so that $e \in \bar{G}$ means $e$ is an edge of $G$. It is now an interesting and suggestive fact that the collection of all paths in $G$ is a category. Thus, it is very far from the case that all categories are very peculiar exotic objects. We first define this basic concept.

**DEFINITION 1.** A category $\mathbf{C}$ consists of:

1. a class $|\mathbf{C}|$ of objects;
2. a class, also denoted $\mathbf{C}$, of morphisms;
3. two functions, $\partial_i : \mathbf{C} \to |\mathbf{C}|$ called source ($i = 0$) and target ($i = 1$), respectively;
4. a function $1 : |\mathbf{C}| \to \mathbf{C}$ assigning to each object $A$ its identity morphism denoted $1_A$, such that $\partial_0 1_A = \partial_0 1_A = A$; and
5. a partial binary operation $\circ$, called composition, defined on a pair $\langle f, g \rangle$ of morphisms iff $\partial_1 g = \partial_0 f$, and then yielding a morphism $f \circ g$, also written $fg$, such that

$$\partial_0(f \circ g) = \partial_0 g \quad \text{and} \quad \partial_1(f \circ g) = \partial_1 f.$$ 

Moreover, the following axioms are required to hold: Composition is associative, in that

$$(f \circ g) \circ h = f \circ (g \circ h)$$

whenever the compositions involved are defined; and $1_A$ satisfies the identity laws whenever their compositions are defined,

$$f \circ 1_A = f \quad \text{and} \quad 1_A \circ g = g.$$ 

For $A, A'$ objects of $\mathbf{C}$, $\mathbf{C}(A, A')$ denotes the class of morphisms in $\mathbf{C}$ such that $\partial_0 f = A$ and $\partial_1 f = A'$. If $f \in \mathbf{C}(A, A')$, we will write $f : A \to A'$. Incidentally, we also use a similar notation for graphs: if $e \in E$ with $\partial_0 e = v$ and $\partial_1 e = v'$, write $e : v \to v'$. Moreover, $f \in \mathbf{C}$ means $f$ is a morphism of $\mathbf{C}$, just as $e \in G$ means $e$ is an edge of $G$.

We will denote the path category of a graph $G$ by $\mathbf{Pa}(G)$. Its objects are the nodes of $G$, i.e., $|\mathbf{Pa}(G)| = |G|$; and its morphisms are the paths in $G$, i.e., the (finite) sequences $e_0 e_1 \ldots e_n$ of adjacent edges in $G$ ("adjacent" means that $\partial_1 e_i = \partial_0 e_{i+1}$, ...
for \( i = 0, \ldots, n - 1 \), including for each node \( v \) of \( G \) an identity or null path \( 1_v \), from \( v \) to \( v \). We shall also find it convenient to write \( G(v, v') \) for \( \mathcal{P}(G)(v, v') \), the set of all paths in \( G \) from \( v \) to \( v' \) for \( v, v' \in |G| \). Composition in \( \mathcal{P}(G) \) is defined to be just ordinary concatenation of sequences, except that there is a confusing reversal of direction: for \( p = e_0 \ldots e_n : v \rightarrow v' \) and \( p' = e_{n+1} \ldots e_{n+m} : v' \rightarrow v'' \), \( p' \circ p = (e_{n+1} \ldots e_{n+m}) \circ (e_0 \ldots e_n) = e_0 \ldots e_n \ldots e_{n+m} \). For this reason we will sometimes let \( \circ \) in \( \mathcal{P}(G) \) stand for concatenation itself, rather than "composition," which is a backwards concatenation. Source and target are defined as follows in \( \mathcal{P}(G) \): \( \partial_0(e_0 \ldots e_n) = \partial_0 e_0 \), and \( \partial_1(e_0 \ldots e_n) = \partial_1 e_n \), with \( \partial_0 1_v = \partial_1 1_v = v \). It is now routine to verify the axioms of Definition 1, so that

**Proposition 1.** \( \mathcal{P}(G) \) is a category.

In fact, the category structure of \( \mathcal{P}(G) \) reflects precisely those aspects of paths which are of greatest general interest, namely, source, target, and composition, with their most important general properties, associativity, identity, and so on. Moreover, \( \mathcal{P}(G) \) is the "free category generated by the graph \( G' \)" in much the same sense that \( X^* \), the monoid of all strings over \( X \) (with concatenation as composition), is the free monoid generated by the set \( X \); for they both satisfy a unique extension property for homomorphisms. In order to make this clear, we need to have available the appropriate notions of homomorphism for graphs and for categories. These will be functions which preserve the algebraic structures involved.

Let \( G = \langle |G|, E, \partial_0, \partial_1 \rangle \) and \( G' = \langle |G'|, E', \partial_0, \partial_1 \rangle \) be graphs. Then a graph morphism \( G \rightarrow G' \) is a pair \( \langle F, F' \rangle \) of functions, \( |F| : |G| \rightarrow |G'| \) taking nodes to nodes, and \( F : E \rightarrow E' \) taking edges to edges, such that the source and target relationships are preserved; that is, such that \( \partial_i(F(e)) = |F|(\partial_i(e)) \) for \( e \in E \), and \( i = 0, 1 \); i.e., such that the diagrams

\[
\begin{array}{ccc}
E & \xrightarrow{F} & E' \\
\downarrow \partial_i & & \downarrow \partial_i \\
|G| & \xrightarrow{|F|} & |G'| \\
\end{array}
\]

commute for \( i = 0, 1 \) (for a diagram to "commute" means that one gets the same resulting function by composing functions along any two paths between the same two objects, here \( E \) and \( |G'| \)). As a standard notation, we shall let \( \langle F, F' \rangle \) be denoted by just \( F \). Given graph homomorphisms \( F : G \rightarrow G' \) and \( F' : G' \rightarrow G'' \), define their composition \( F'F : G \rightarrow G'' \) to be \( \langle F'| |F|, F'F \rangle \), the pair of function compositions. The notion of a homomorphism of categories, called a functor, is really quite similar, except that the additional structural features of composition and identities must be preserved.
DEFINITION 2. A functor $F$ from a category $B$ to a category $C$, written $F : B \to C$, consists of an object part, a function $|F| : |B| \to |C|$, and a morphism part, a function $F : B \to C$, such that $\delta_i(F) = |F|(|\delta_i f|) = \delta_i(F) = IF \cdot (Fg)$, whenever $f \circ g$ is defined in $B$; and $F(1_A) = 1_{|F| A}$ for all objects $A$ of $B$. For objects $A$, $A'$ of $B$, we let $F_{AA'}$ denote the morphism part of $F$ restricted to $B(A, A')$; thus $F_{AA'} : B(A, A') \to C(FA, FA')$. Given functors $F : B \to C$ and $F' : A \to B$, define their composition $FF'$ : $A \to C$ by $|FF'| |(A) = |F|(|F'| |(A))$ for $A \in |A|$, and $FF'(f) = F(F'(f))$ for $f$ a morphism of $A$ (it is easy to see $FF'$ is also a functor).

The structural relationship between categories and graphs is quite close. In fact, any category $C$ can be "viewed as" a graph by forgetting about its composition and its association of identities to objects. This "underlying graph" of $C$ will be denoted $V(C)$ in this paper (perhaps for "vergessen", as $V$ "forgets" some structure). Similarly, any functor $F : B \to C$ can be "viewed as" a graph morphism $V(F) : V(B) \to V(C)$ between the underlying graphs of its source and target categories. Notice in particular that the inclusion $i_G : G \to V(Pa(G))$ of a graph into the underlying graph of its path category, defined by $|i_G|(|v|) = |v|$ and $i_G(|e|) = |e|$ (as a path of length one), is a graph morphism. We now show that this inclusion is "universal," i.e., that through it $G$ freely generates $Pa(G)$ in essentially the same sense that set $X$ freely generates the monoid $X^*$.

THEOREM 2. Given a graph $G$ and a category $C$, every graph morphism $P : G \to V(C)$ has a unique extension to a functor $\hat{P} : Pa(G) \to C$ (this means that there is a unique $\hat{P}$ such that $V(\hat{P}) \circ i_G = P$, where $\circ$ indicates composition of graph morphisms). Conversely, every functor $Pa(G) \to C$ is of the form $\hat{P}$ for a unique graph morphism $P : G \to V(C)$.

The following diagram may be helpful in visualizing this result.

```
```

Proof. For $v \in |G|$ we must have $|P| | v | = | V(\hat{P}) \circ i_G| | v | = | V(\hat{P})| | v | = | \hat{P} | | v |$; and for $e \in E$ we must have $P e = V(\hat{P}) \circ i_G(e) = V(\hat{P})(e) = \hat{P}(e)$. Moreover, $\hat{P}$ must be a functor, so we must have $\hat{P}(1_v) = 1_{|P| v} = 1_{|P| v}$; and if $f = e_0 \ldots e_k : v \to v'$ is a path in $G$, then $\hat{P}(f) = \hat{P}(e_0 \ldots e_k)$ must equal $\hat{P}(e_0) \circ \ldots \circ \hat{P}(e_k) = P(e_0) \circ \ldots \circ P(e_k)$. One now easily checks that these necessary conditions in fact constitute the definition of a functor $\hat{P}$; and of course the above necessity argument gives the desired uniqueness.

1 Double underscore in diagrams is equivalent to boldface type in text.
Given a functor $F : \text{Pa}(G) \to C$, $V(F) : V(\text{Pa}(G)) \to V(C)$ and $V(F) \circ i_G : G \to V(C)$ are graph morphisms; let the latter be $P$. Then there is a unique $\hat{P}$ such that $V(\hat{P}) \circ i_G = P$. Since $F$ satisfies this condition by definition, we have $\hat{P} = F$. Q.E.D.

Notice that the extension of $P$ to $\hat{P}$ proceeds the same way as for a function $f : X \to M$ to a monoid, by composition; for $f : X^* \to M$ is given by $f(x_0 \ldots x_n) = f(x_0) \circ \ldots \circ f(x_n)$, where $\circ$ is the monoid operation in $M$. Hereafter we shall feel free to write $C$ or $\text{Pa}(G)$ when we mean $V(C)$ or $V(\text{Pa}(G))$, as is also parallel to the usual convention for monoids (i.e., $X^*$ may denote either the set of strings, or the monoid of strings); context should make the intended meaning clear.

It follows from Theorem 2 that if $G$ is finite, then every functor $F : \text{Pa}(G) \to C$ is determined by a finite amount of data, even though $\text{Pa}(G)$ itself may be quite infinite. For the (unique) graph morphism $P : G \to C$ such that $P \circ F$ is described by two finite tables, giving the values which $P$ assigns to the nodes, and to the edges, of $G$. We will show in the following sections that this observation lies behind the effective (i.e., finitely computable) character of Floyd’s method. In fact, the freeness, or universal property, of $\text{Pa}(G)$ amounts to an algebraic formulation of an inductive principle of definition.

Some of our later applications of the path category require certain additional properties, which we now exposit. However, the reader may quite profitably skip to the next section of this paper, and return here only when he feels a need for it.

If $F : G_0 \to G_1$ is a graph morphism, we define its path extension, $\hat{F} : \text{Pa}(G_0) \to \text{Pa}(G_1)$ to be the extension to $\text{Pa}(G_0)$ given by Theorem 2 of the composite graph morphism $G_0 \circ F \Rightarrow \text{Pa}(G_1)$, where $i_1$ stands for $i_{G_1}$; i.e., $\hat{F} = (i_1 \circ F)^\wedge$.

Thus, $\hat{F}$ is the unique functor $\text{Pa}(G_0) \to \text{Pa}(G_1)$ such that $i_0 \circ V(F) = i_1 \circ F$. Now looking at the construction of the extension given in the proof of Theorem 2, we see that $v \in \text{Pa}(G_0)$, $F(v) = (i_1 \circ F)^\wedge (v) = i_1(F(v)) = F(v)$; and for $f = e_0 \ldots e_k : v \to v'$ a path in $G_0$, $F(f) = (i_1 \circ F)^\wedge (f) = i_1(F(e_0)) \circ \ldots \circ i_1(F(e_k)) = F(e_0) \ldots F(e_k)$. In a sense, this construction of $\hat{F}$ from $F$ defines $\text{Pa}$ itself as a functor from the category of graphs to that of categories; thus, were it not so awkward, $\text{Pa}(F)$ would be a reasonable notation for $\hat{F}$. The basic functorial property of this construction and an additional fact which will be needed in Section 7, are given in the following result.
Proposition 3. Let $F_0 : G_0 \rightarrow G_1$ and $P : G_1 \rightarrow V(C)$ be graph morphisms, for $C$ a category. Then $(PF_0)^\wedge = \bar{P}F_0$. Let $F_0 : G_0 \rightarrow G_1$ and $F_1 : G_1 \rightarrow G_2$ be graph morphisms. Then $(F_1F_0)^\wedge = F_1^\wedge F_0^\wedge$.

Proof. The uniqueness assertion of Theorem 2 and the following diagram can be used to give the first assertion.

\[
\begin{align*}
G_0 & \xrightarrow{F_0} G_1 & \xrightarrow{P} V(C) \\
\downarrow i_0 & \downarrow i_1 & \\
V\text{Pa}(G_0) & \xrightarrow{V(F_0)} V\text{Pa}(G_1) & \xrightarrow{V(P)} V(C)
\end{align*}
\]

The second assertion follows from the first by substituting $i_0 \circ F_1$ for $P$ and $\text{Pa}(G_0)$ for $C$. Q.E.D.

3. Flow Diagram Programs

Speaking heuristically now, in this section we shall consider programs consisting of operations and tests, each performed directly on values stored in memory. Thus we are considering “low level” code, such as might be the output of a compiler; for the kind of algebraic considerations needed to treat the semantics of higher level languages, see [18]. Operations are transformations defined on memory “states” or “values,” and tests produce choices among alternative paths of execution of the operations. Operations are represented by functions describing the changes induced on states, and tests are represented by partial identity functions, producing no state change, but defined only on those values (states) where the test “succeeds” (actually, tests and operations can be combined, in a form represented by partial nonidentity functions). For example, the operation “$X = X + 1$” on nonnegative integers is represented by the function $\omega \mapsto \omega$ defined by $x \mapsto x + 1$; here $\omega$ represents the set \{0, 1, 2,...\} of all nonnegative integers, and the arrow “$\mapsto$” is read “goes to.” The test “$X > 0$” on nonnegative integers is represented by the partial identity function $\omega \mapsto \omega$, defined only on the positive integers; similarly “$X = 0$” is the partial identity function defined on $\omega$ only at zero. This use of partial subfunctions of the identity seems to go back to Karp [11]. These tests and operations will appear as (labels of) edges in a graph, with all of the partial functions representing the several alternatives of a test emanating from the same node. Thus, a path in this graph represents an execution sequence for instructions of the program; otherwise put, a path represents a flow of executive control in the program. It should be noted that these flow diagram programs (which will soon be defined quite precisely) are not purely syntactic entities: a specific interpretation is assumed to be already given for each operation and test instruction. The questions of greatest interest for such a program are semantic: does it always terminate? What function does it compute? And so on.
The rigorous algebraic approach we shall take to the theory of flow diagram programs requires some categories which are much larger than the path categories. Let $\mathbf{Pfn}$ be the category with (all possible) sets as objects, with (all possible) partial functions $f : A \to B$ as morphisms, with $\delta_0(f : A \to B) = A$ and $\delta_1(f : A \to B) = B$, with $1_A$ the identity function $A \to A$, and with composition the usual (Pierce product) composition of partial functions viewed as relations. Thus, given $f : A \to B$ and $g : B \to C$ partial functions, $g f : A \to C$ is defined on an $a \in A$, with value $g(f(a))$, if $f$ is defined on $a$ and $g$ is defined on $f(a)$; we shall call the subset of its source on which a partial function is actually defined, its set of definition. The reader can readily verify that $\mathbf{Pfn}$ in fact satisfies the axioms of Definition 1. Similarly, we let $\mathbf{Rel}$ denote the category with sets as objects, relations $f : A \to B$ as morphisms, and with source, target, identities and composition as in $\mathbf{Pfn}$. Call $\mathbf{Rel}$ the category of relations, and $\mathbf{Pfn}$ the category of partial functions. We are now in a position to give this section's basic concept.

**Definition 3.** A program is a functor $P : \mathbf{Pa}(G) \to \mathbf{Pfn}$, for some graph $G$. A deterministic program is a program $P$ such that whenever $e$ and $e'$ are edges with the same source node, the partial functions $P e$ and $P e'$ have disjoint sets of definition. A nondeterministic program is a functor $\mathbf{Pa}(G) \to \mathbf{Rel}$. For any (possibly nondeterministic) program $P$, call $G$ the underlying graph or shape of $P$, and let $| P |$ stand for $| G |$, the set of vertices underlying $P$.

Generally in the following, “program” can be understood as either deterministic or nondeterministic; if the distinction is particularly important, we will prefix the appropriate modifier.

Thus, by Theorem 2 a program is an assignment on a graph of sets to nodes and partial functions to edges, then extended by composition to paths. Elements of the sets assigned to nodes correspond to what McCarthy called program state vectors, while the partial functions assigned to edges represent the computational steps, including tests for branches, as already described.

We now give a simple example, whose semantics we shall explore later. Let $G$ be the graph with node set $\{ a, b, c, d, e \}$ and edges as indicated schematically below:

Since this $G$ never has more than one edge $v \to v'$, for fixed nodes, $v, v'$, it will be unambiguous to denote the edge from $v$ to $v'$ (if there is one) by the pair $\langle v, v' \rangle$. Otherwise put, the function $\langle \delta_0, \delta_1 \rangle : E_1 \to | G | \times | G |$ (sending $e$ to $\langle \delta_0 e, \delta_1 e \rangle$) is injective.
It will suffice to define a program $P$ on nodes and edges, the extension to paths being direct by Theorem 2 (see later for further explanation of this point). Let $P(a) = \omega^2$ (the set of pairs of nonnegative integers), $P(b) = \omega^3$ (triples from $\omega$), $P(c) = \omega^3$, $P(d) = \omega^3$ and $P(e) = \omega$. We shall denote the vectors in these sets by components as $\langle X, Y \rangle$, $\langle X, Y, Z \rangle$, and $\langle Z \rangle = Z$ respectively. Now let partial functions be assigned to the edges as below:

Using the pair notation, we can give precise definitions of the functions on edges as follows. $P(\langle a, b \rangle): \omega^2 \rightarrow \omega^3$ by $\langle X, Y \rangle \mapsto \langle X, Y, 0 \rangle$ (\rightarrow is read “goes to”); $P(\langle b, c \rangle): \omega^3 \rightarrow \omega^3$ is the partial identity function defined only if the second component is positive; $P(\langle c, d \rangle): \omega^3 \rightarrow \omega^3$ by $\langle X, Y, Z \rangle \mapsto \langle X, Y, 2Z + X \rangle$; $P(\langle d, b \rangle): \omega^3 \rightarrow \omega^3$ by $\langle X, Y, Z \rangle \mapsto \langle X, Y - 1, Z \rangle$ defined only if $Y > 0$; and, $P(\langle b, e \rangle): \omega^3 \rightarrow \omega$ by $\langle X, Y, Z \rangle \mapsto Z$, but defined only if $Y = 0$. We will give more complicated examples later, but from now on we shall prefer to use the simple schematic representation of the immediately preceding diagram rather than detailed set theoretic descriptions of the node and edge sets.

The nodes of the underlying graph $G$ of a program represent the “control states” of the program (e.g., the line of code about to be executed), so that paths in $Pa(G)$ represent the possible “flows of control” during execution of the program. Note that defining $P$ on these five edges determines $\tilde{P}$ on an infinite number of paths, the iterations around the loop (or cycle) in $G$, by composition.

By Theorem 2, a program $P: Pa(G) \rightarrow Pf\tilde{n}$ is completely determined by the graph morphism $V(P) \circ i_G: G \rightarrow V(Pf\tilde{n})$, which can be finitely described if $G$ is finite. This constitutes an assignment of sets to nodes and partial functions to edges, as in the example above. In the following we generally think of a program as being so defined and will use the symbol $P$ for the graph morphism. Of course, $\tilde{P}$ will always denote the functorial extension of $P$, but sometimes we also let $P$ denote a functor $Pa(G) \rightarrow Pf\tilde{n}$. These remarks apply to nondeterministic programs, with $Pf\tilde{n}$ replaced by $Rel$.

4. PROGRAM HOMOMORPHISMS

We now turn to the main concept of this paper, program homomorphism. This concept is fundamental to our techniques for proving correctness, termination, and equivalence, as given in later sections. We first recall another basic definition from category theory.
DEFINITION 4. A natural transformation $\eta : G \Rightarrow H$ from a functor $G$ to a functor $H$, both with source category $A$ and target category $B$, is a family $\eta_v : Gv \to Hv$ of morphisms in $B$, one for each object $v$ in $A$, such that for each morphism $f : v \to v'$ in $A$, the following diagram commutes in $B$

\[
\begin{array}{ccc}
Gv & \xrightarrow{\eta_v} & Hv \\
\downarrow{g} & & \downarrow{h} \\
Gv' & \xrightarrow{\eta_{v'}} & Hv'
\end{array}
\]

A program homomorphism $P_0 \to P_1$ will have two parts: one mapping paths, or flows of control in $P_0$ (consistently) to other paths in $P_1$; the other expressing consistency, relative to the flow of control mapping, of operations in $P_0$ with other operations in $P_1$, via a natural transformation.

DEFINITION 5. Let $\hat{P}_0 : Pa(G_0) \to Pfn$ and $\hat{P}_1 : Pa(G_1) \to Pfn$ be programs. Then a homomorphism $\hat{P}_0 \to \hat{P}_1$ (also written $P_0 \to P_1$) is a pair $\langle F, \eta \rangle$, where $F : Pa(G_0) \to Pa(G_1)$ is a functor, and $\eta : P_0 \Rightarrow P_1 \circ F$ is a natural transformation, with each $\eta_v$ a partial function. A simulation of $P_0$ by $P_1$ is a homomorphism $P_0 \to P_1$ such that each value $\eta_v$ of $\eta$ (for $v \in \mid G_0 \mid$) is an inclusion. A projection of $P_0$ to $P_1$ is a homomorphism $P_0 \to P_1$ such that $F$ and each $\eta_v : P_0v \to P_1Fv$ are total surjective.

The following diagram may help visualize the situation of Definition 5.

\[
\begin{array}{ccc}
Pa(G_0) & \xrightarrow{F} & Pa(G_1) \\
\downarrow{\hat{P}_0} & & \downarrow{\hat{P}_1} \\
\text{Pfn} & \xrightarrow{\eta} & \text{Pfn}
\end{array}
\]

Note that $\hat{P}_1 \circ F$ is the composition of functors.

We defer giving examples until we come to the correctness and other proofs for which these concepts are intended. It should be noted that exactly the same definition applies to homomorphisms of nondeterministic programs: just replace $Pfn$ by $Rel$. The intuitive meaning of Definition 5 is that $\langle F, \eta \rangle : P_0 \to P_1$ maps paths, or flow of control, in $P_0$ (consistently) to other paths in $P_1$, via $F$; and maps the operations performed along these paths correspondingly via $\eta$. Of course, $\eta$ does not really map operations at all, but its naturality does impose a consistency constraint upon the operations along paths in $G_1$ relative to the corresponding operations along paths in $G_0$. In particular, if each $\eta_v$ is an inclusion, the operation $P_1(F(e))$ must equal $P_0(e)$ on the set of definition of $P_0(e)$, i.e., $P_0(e)$ is a subfunction of $P_1(F(e))$. Incidentally, it would also be quite reasonable to define a simulation to have each $\eta_v$ injective rather an inclusion; this would permit one to use different symbols in the simulating
and simulated program's semantics. However, it is not desirable to put any restrictions at all on $F$. Burstall [1] (and Milner [15]) in effect require $F$ to be an identity or isomorphism; the present definition allows one to contract and expand paths, and to fold large (even unbounded) iterations into loops. The very interesting recent work of R. Burstall [2] in which morphisms are sets of paths, should also be mentioned; it provides a way of replacing single statements by programs which may even include loops.

The principal conceptual motivation for Definition 5 is that a program as defined above is a system in the precise sense of Goguen [5, 6] so that it is also natural to use the corresponding notion of morphism in the new context. That is, the definition of program homomorphism is a special case of that of a general system morphism (note however that this involves a reversal of the direction of the natural transformation). There is also a suitable notion of composition for homomorphisms of programs, again following Goguen [5] for systems, and this gives a category Prog whose objects are programs and whose morphisms are program homomorphisms. As this notion is rather complicated and is used only in Section 7 of this paper, we defer it to that point.

The following result is the basis for verifying in a finite number of steps that one has a program homomorphism.

**Theorem 4.** For $\langle F, \eta \rangle : P_0 \rightarrow P_1$ to be a program homomorphism, it is necessary and sufficient that: (1) $F$ is the extension $\hat{H}$ of a graph morphism $H : G_0 \rightarrow \mathbf{Pa}(G_1)$; and (2) for each edge $e : v \rightarrow v'$ in $G_0$, the diagram

$$
\begin{array}{ccc}
P_0v & \xrightarrow{\eta_v} & P_1(Fv) \\
\downarrow_{F_0e} & & \downarrow_{P_1(Fe)} \\
P_0v' & \xrightarrow{\eta_{v'}} & P_1(Fv')
\end{array}
$$

commutes in $\mathbf{Pfn}$.

**Proof.** From Theorem 2 we already know that $F : \mathbf{Pa}(G_0) \rightarrow \mathbf{Pa}(G_1)$ is a functor iff it is the extension $\hat{H}$ of a graph morphism $H : G_0 \rightarrow V(\mathbf{Pa}(G_1))$. This disposes of condition (1). We now use a trick in order to dispose of condition (2) in exactly the same way. In fact, there is a bijection between on the one hand natural transformations of functors $\mathbf{Pa}(G_0) \rightarrow \mathbf{Pfn}$, and on the other hand functors $\mathbf{Pa}(G_0) \rightarrow \square \mathbf{Pfn}$, where $\square \mathbf{Pfn}$ is the category whose objects are partial functions, and whose morphisms are commutative squares of partial functions. For example, if $f, g$ and $h$ are partial functions, morphisms $f \rightarrow g$ and $g \rightarrow h$ in $\square \mathbf{Pfn}$ are commuting squares of the forms

$$
\begin{array}{ccc}
a & \xrightarrow{f} & a' \\
\downarrow & & \downarrow \\
b & \xrightarrow{\;} & b'
\end{array}
$$

and

$$
\begin{array}{ccc}
a' & \xrightarrow{g} & a \\
\downarrow & & \downarrow \\
b' & \xrightarrow{h} & b
\end{array}
$$
respectively. Composition arises by "pasting together" squares along their common side.

Then given $\eta : F \Rightarrow G$, for $F, G : \text{Pa}(G_0) \rightarrow \text{Pfn}$, define $\eta \Box : \text{Pa}(G_0) \rightarrow \Box \text{Pfn}$ by $\eta \Box(v) = \eta_v : Fv \rightarrow Gv$, a morphism in $\text{Pfn}$ and object in $\Box \text{Pfn}$; and for $e : v \rightarrow v'$ in $\text{Pa}(G_0)$, $\eta \Box(e)$ is defined to be the square

$$
\begin{array}{ccc}
Fv & \xrightarrow{Fe} & Fv' \\
\downarrow{\eta_v} & & \downarrow{\eta_{v'}} \\
Gv & \xrightarrow{Ge} & Gv'
\end{array}
$$

in $\text{Pfn}$, a morphism $\eta \Box(v) \rightarrow \eta \Box(v')$ in $\text{Pfn}$.

We now apply Theorem 2 to the functor $\text{Pa}(G_0) \rightarrow \Box \text{Pfn}$ corresponding to the natural transformation $\eta$, to see that a graph morphism $G_0 \rightarrow V(\Box \text{Pfn})$ will suffice to determine (or describe) $\eta$. But such a graph morphism is just what is given in condition (2). Q.E.D.

By this result, a program homomorphism can be, and in the following generally will be, described by a graph morphism $H : G_0 \rightarrow \text{Pa}(G_1)$ and a collection of partial functions $\eta_v$ satisfying the condition (2). If $G_0$ is finite, then the family of $\eta_v$'s is finite, the condition (2) is finitely verifiable (one diagram for each edge), and by a previous remark $H$ is also finitely describable.

Note that the corresponding result for nondeterministic programs is stated and proved in the same way, replacing $\text{Pfn}$ by $\text{Rel}$.

5. The Behavior of Programs

If $f : v \rightarrow v'$ is a path in $G$ and $\hat{P} : \text{Pa}(G) \rightarrow \text{Pfn}$ is a program, then $\hat{P}(f) : Pv \rightarrow Pv'$ is the partial function computed by $P$ if it happens to go along that path; of course, $\hat{P}(f)$ can be the empty function. The same holds for a nondeterministic program $\hat{P} : \text{Pa}(G) \rightarrow \text{Rel}$. For deterministic programs, there is an important relationship between the functions computed along paths with the same source.

**Proposition 5.** If $P$ is deterministic, if $v$ is a node of $G$, and if $f, f'$ are paths with source $v$ such that neither is an initial segment of the other, then $\hat{P}(f)$ and $\hat{P}(f')$ have disjoint sets of definition.

**Proof.** Write $f = geh$, $f' = ge'h'$, with edges $e \neq e'$ (and $g$ possibly the identity path at $v$). Then the sets of definition $\text{def}(f)$, $\text{def}(f')$ of $f, f'$ are contained in the sets $\hat{P}(h)^{-1}(\text{def}(P(e)))$, $\hat{P}(h')^{-1}(\text{def}(P(e')))$, respectively, which are disjoint since $\text{def}(Pe)$, $\text{def}(Pe')$ are disjoint by the hypothesis that $P$ is deterministic. Q.E.D.
It follows that if \( v' \) is an exit node, in the sense that there are no edges in \( G \) with source \( v' \), then for a deterministic program \( P \) and fixed \( v \), the paths \( v \rightarrow v' \) in \( G \) give rise to pairwise disjoint partial functions \( P_v \rightarrow P_{v'} \), each representing a possible termination \( v' \) from an entry at \( v \). Thus we have

**Definition 6.** The behavior of, or complete partial function computed by \( \hat{P} : \text{Pa}(G) \rightarrow \text{Pfn} \), with entry at \( v \) and exit at \( v' \) (in a slightly loose notation) is

\[
\bigcup \{ \hat{P}(f) \mid f : v \rightarrow v' \text{ in } \text{Pa}(G) \},
\]

which will hereafter be denoted \( P(v, v') \). More generally, if \( X \) and \( A \) are sets of nodes of \( G \), define \( P(v, X) = \bigcup \{ P(v, v') \mid v' \in X \} \), corresponding to exit at any node in \( X \) and \( P(A, X) = \bigcup \{ P(v, v') \mid v \in A \text{ and } v' \in X \} \), corresponding to entry from any node in \( A \) and exit from any node in \( X \).

The union symbol is meant in the sense that each \( \hat{P}(f) \) is thought of as a set of ordered pairs (a subset of \( P_v \times P_{v'} \)). The looseness comes from the fact that (at least in categorical contexts) functions should be thought of as ordered triples \( \langle S, R, T \rangle \), where \( S \) is the source set, \( T \) the target set, and \( R \) the set of ordered pairs; in fact, an “inclusion function” cannot be distinguished from the identity function (on the source) without explicit knowledge of the target set; and partial functions cannot be distinguished from total functions without explicit knowledge of their sources.

Proposition 5 and the above remarks give the following.

**Corollary 6.** If \( P \) is deterministic and \( v' \) is an exit node, then \( P(v, v') \) is also a partial function, rather than a relation. More generally, if \( X \) is a set of exit nodes, \( P(v, X) \) is a partial function.

If \( P \) is just a program, or even nondeterministic, then the formula for \( P(v, v') \) still makes sense, but it may very well define a relation which is not a partial function. The formula also makes sense when \( v' \) is not an exit node, and we shall freely use the same notation \( P(v, v') \) for the relation arising in any of these cases. The relation-valued function \( P(\_ , \_ ) \) defined on pairs of nodes describes the semantics, meaning, or behavior of the program \( P \). Often programs are set up so that only one particular relation \( P(v, v') \) is of direct interest, with \( v \) an entrance and \( v' \) an exit node. For the program \( P \) given in Section 3, we are interested in \( P(a, e) \) as a relation \( \omega^2 \rightarrow \omega \). The next section gives techniques based on Section 4, which will be used to compute \( P(a, e) \) for this example.

It might be remarked parenthetically that, from the point of view of the general system theory of [5, 6], the notion of behavior given in this section is somewhat oversimplified and should be modified in various ways.
6. CORRECTNESS AND TERMINATION

We now give the results which will shortly be used in sample correctness proofs. These results repeatedly use the following set theoretic fact.

**Lemma 7.** Let $R : A \rightarrow B$, $S : B \rightarrow C$, $R_i : A \rightarrow B$, and $S_i : B \rightarrow C$ for $i \in I$ (some index set) be relations. Then

$$\bigcup_i (S_i \circ R) = (\bigcup_i S_i) \circ R \quad \text{and} \quad \bigcup_i (S \circ R_i) = S \circ (\bigcup_i R_i),$$

where $\circ$ denotes the (Pierce product) composition of relations.

**Proposition 8.** If $\langle F, \eta \rangle : P_0 \rightarrow P$ is a program homomorphism such that for nodes $v, v'$ of $G_0$, both $\eta_v$ and $\eta_{v'}$ are inclusions, then

$$P_0(v, v') \subseteq P(Fv, Fv').$$

**Proof.** Let $f : v \rightarrow v'$ be a path in $G_0$. Then by naturality of $\eta$ we have commutativity of

\[
\begin{array}{ccc}
\text{P}_0(v) & \xrightarrow{\eta_v} & \text{P}(Fv) \\
\text{P}_0(v') & \xrightarrow{\eta_{v'}} & \text{P}(Fv')
\end{array}
\]

with the horizontal arrows inclusions by hypothesis. This implies that $\hat{P}_0 f$ is a subfunction of $\hat{P}Ff$ (which we write as $\hat{P}_0 f \subseteq \hat{P}Ff$). Then each function in the union $P_0(v, v') = \bigcup \{ \hat{P}_0 f : v \rightarrow v' \text{ in } \text{Pa}(G_0) \}$ is a subfunction of a function in the union $P(Fv, Fv') = \bigcup \{ \hat{P}g : Fv \rightarrow Fv' \text{ in } \text{Pa}(G) \}$, since $Ff : Fv \rightarrow Fv'$ is in $\text{Pa}(G)$ whenever $f : v \rightarrow v'$ in $\text{Pa}(G_0)$. Thus (one can use Lemma 7 here, with $R = \eta_{v'}$, $S = \eta_v$) $P_0(v, v')$ is a subrelation of $P(Fv, Fv')$. Q.E.D.

The following immediate consequence of this result helps justify our use of the word "simulation" for the concept given in Definition 5.

**Corollary 9.** If $\langle F, \eta \rangle : P_0 \rightarrow P$ is a simulation, then for all nodes $v, v'$ of $P_0$, $P_0(v, v') \subseteq P(Fv, Fv')$.

That is, for a simulation $P_0 \rightarrow P$, whatever $P_0$ computes, from any $v$ to any $v'$, $P$ can also compute with corresponding entry and exit nodes $Fv$ and $Fv'$ (Proposition 8
makes the same assertion with a fixed $v$ and $v'$). Both these results apply directly
to nondeterministic programs and easily extend to sets of entry and exit nodes.

**Proposition 10.** Let $\langle F, \eta \rangle : P_0 \rightarrow P$ be a (nondeterministic) program homomorphism
and let $A, X \subseteq |P_0|$ such that for all $v \in A \cup X$, $\eta_v$ is an inclusion. Then

$$P_0(A, X) \subseteq P(FA, FX).$$

**Proof.** For each $v \in A$ and $v' \in X$ we have $P_0(v, v') \subseteq P(Fv, Fv')$ by Proposition 8.
Therefore

$$P_0(A, X) = \bigcup_{v, v'} P_0(v, v') \subseteq \bigcup_{v, v'} P(Fv, Fv') = P(FA, FX).$$

Q.E.D.

One convenient method for proving correctness of a program $P$ is as follows.

Find a program $P_0$ simulated by $P$ such that the sets assigned (or "attached") to
nodes of $P_0$ reflect the relationships to be proved for $P$. These sets at nodes of $P_0$
correspond to Floyd’s “assertions” for $P$. Assume that sets $A, A'$ are attached to
nodes $v, v'$ of a deterministic program $P_0$ (i.e., $P_0 v = A, P_0 v' = A'$),
and assume without loss of generality that $P_0(v, v') : A \rightarrow A'$ is a total function (otherwise replace
$A$ by the set of definition of $P_0(v, v')$). Then if $\langle F, \eta \rangle$ is a simulation,
$P_0(v, v')$ is a
subfunction of the (partial) function $P(Fv, Fv')$ computed by deterministic $P$. This
means that if the data of $P$ satisfies $A$ upon entry at $Fv$, then it satisfies $A'$ upon exit
at $Fv'$; or equivalently, that $P(Fv, Fv')(A) \subseteq A'$. The value of Theorem 4 is that if $P_0$
is finite we can check the necessary hypothesis, that $\langle F, \eta \rangle$ is a homomorphism, in a
finite number of steps. This method applies even if the programs have different shapes,
although of course the shapes cannot be too radically different because of the require-
ment that flow of control be "preserved."

Perhaps the most convenient special case it that $A$ is a singleton set containing a
"generic" or "typical" data point, and $A'$ is a relation we want to show is satisfied
by programs entering with this point as data, for example, a formula with the point
as argument. We now use this technique to show correctness of the program given as
an example in Section 3, which we shall still call $P$. Let $P_0$ be the program indicated
schematically by

$$\begin{align*}
\{\langle x, y \rangle \} \\
Z := 0 \\
\{\langle x, i, (2^{y-1}-1) \cdot x \rangle | 0 \leq i \leq y \} \\
Y > 0 \\
Y := Y - 1 \\
Z := 2Z + X \\
\{\langle 2^Y - 1 \rangle \cdot x \} \\
\end{align*}$$
This program has only three nodes, call them $a$, $b$, and $e$ from top to bottom, and
three edges, denoted $\langle a, b \rangle$, $\langle b, b \rangle$, and $\langle b, e \rangle$, which compute the (partial) functions
$\langle X, Y \rangle \mapsto \langle X, Y, 0 \rangle$, $\langle X, Y, Z \rangle \mapsto \langle X, Y - 1, 2Z + X \rangle$ if $Y > 0$, and $\langle X, Y, Z \rangle \mapsto Z$ if $Y = 0$, respectively. The assignment of sets to nodes is evident from the picture,
but one also has to check that the assignments always give values in their target sets
from argument their source sets. Thus, along $\langle a, b \rangle, \langle x, y \rangle$ goes to $\langle x, y, 0 \rangle$ which is
of the form $\langle x, i, (2^y - 1)x \rangle$ for $i = y$; along $\langle b, b \rangle$ for $i > 0$, $\langle x, i, (2^y - 1)x \rangle$
go to $\langle x, i - 1, (2^y - 1)x \rangle$, again of the same form; and along $\langle b, e \rangle$ if $i = 0$,
$\langle x, i, (2^y - 1)x \rangle$ goes to $(2^y - 1)x$, as desired. Let $H : G \to \text{Pa}(G)$ take the
"entering" and "exiting" edges $\langle a, b \rangle$ and $\langle b, e \rangle$ in $G$ to their correspondents in $G$,
and take the "loop" edge $\langle b, b \rangle$ in $G$ to the path $\langle b, c, d, b \rangle$ in $G$. This is easily seen
to be a graph morphism, and using Theorem 2 we let $F = \hat{H} : \text{Pa}(G) \to \text{Pa}(G)$. Let
each $\eta_\nu$ for $\nu \in \{a, b, e\}$ be the evident inclusion. Using Theorem 4, we have now to
check commutative squares, which amount to three subfunction relationships; in fact,
these hold by the construction of $P_0$. Notice that this provides an example of a
homomorphism of programs which expands edges to paths. We conclude, now using
Proposition 8 (or Corollary 9) that $P_0(a, e) \subseteq P(a, e)$. But $P_0(a, e) \subseteq P_0(a) \times P_0(e) =
\{ \langle x, y \rangle, (2^y - 1) \cdot x \}$, so that either $P_0(a, e) \langle x, y \rangle$ is undefined, or is $(2^y - 1)x$.
This is valid for any $\langle x, y \rangle \in \omega^2$. Therefore, if $P_0$ terminates at $e$ from $a$, in the sense
that $P_0(a, e)$ is never undefined, it must be the function $P_0(a, e) \langle x, y \rangle = (2^y - 1) \cdot x$.
Since $P$ is deterministic and $e$ is an exit node, $P(a, e)$ is a (partial) function by Corollary
5. But assuming termination, we have a pair in $P(a, e)$ for each point $\langle x, y \rangle$ in its
source, so that $P(a, e) \langle x, y \rangle = (2^y - 1)x$, for all $x, y \in \omega$.

If we had written $P$ specifically to compute $(2^y - 1)x$, the above would be a "partial
correctness proof"; otherwise, it is a "partial" determination of the semantics or
behavior of $P$. "Partial" refers to the need to prove termination separately, as with most
other methods. It might also be desirable to show a program $P$ terminates, without the
extra effort of showing exactly what the program semantics is. Either way, we can use
a program homomorphism $P \to P_1$ which preserves termination but possibly destroys
many detailed computational properties of $P$. Showing termination means showing
some function is total, as was required for $P(a, e)$ in the above discussion. Thus,

Definition 7. A program (or nondeterministic program) $P$ terminates at $v'$ from
$v$ iff $P(v, v')$ is total (i.e., its set of definition equals its source). Similarly, a program $P$
terminates at $X \subseteq |P|$ from $v \in |P|$ iff $P(v, X)$ is total; and at $X \subseteq |P|$ from $A \subseteq |P|$ iff $P(A, X)$ is total.

For nondeterministic programs this means only that there is some path along which
each entering data point leads to termination, although it may stagnate along other
paths; but in the deterministic case this cannot happen, since by Corollary 5 there is
at most one path to termination. The following result justifies the termination verifica-

Method we are suggesting.
Proposition 11. If \( \langle F, \eta \rangle : P \rightarrow P_1 \) is a program homomorphism such that \( F_{v'} \) is surjective and \( \eta_v \) is total, and if \( P_1 \) terminates at \( F_{v'} \) from \( F_v \), then \( P \) terminates at \( v' \) from \( v \). Similarly, for any \( F, P_1 \) terminates at \( v' \) from \( v \) implies \( P \) terminates at \( F^{-1}v' \) from \( F^{-1}v \), provided these sets are nonempty.

The proof is similar to that of Proposition 8.

Notice that \( \eta_v \) and \( \eta_{v'} \) do not have to be surjective or injective, and that indeed there are no conditions at all on \( \eta_{v'} \). The conditions on \( F \) are needed to insure that termination in \( P_1 \) does not occur along a path unavailable to \( P \). Totality of \( \eta_v \) is hardly any restriction at all in our applications, since \( \eta_v \) could generally be extended to all of \( P_v \), possibly also extending \( PF_v \). In fact, \( \langle F, \eta \rangle : P \rightarrow P_1 \) is often a projection in termination proofs, and since a projection satisfies the hypotheses of Proposition 11 for any \( v, v' \), we have the following.

Corollary 12. If \( P \rightarrow P_1 \) is a projection, then for all \( v, v' \) vertices of the underlying graph of \( P_1 \), \( P \) terminates at \( F^{-1}v \) from \( F^{-1}v' \) whenever \( P_1 \) terminates at \( v \) from \( v' \).

We now use this method to prove termination of the program \( P \) of Section 3, by creating a crude model \( P_1 \) of \( P \), which keeps track of the flow of control by decrementing a single integer valued register \( Y \), and otherwise does no calculation at all.

Let the nodes of \( P_1 \) be \( a, b, c, e \) from top to bottom as before, and let its sets and functions be as indicated above. Define a graph morphism \( H : G \rightarrow \text{Pa}(G_1) \) by:

\[
\langle a, b \rangle \mapsto \langle a, b \rangle, \quad \langle b, c \rangle \mapsto \langle b, c \rangle, \quad \langle c, d \rangle \mapsto 1_c, \quad \langle d, b \rangle \mapsto \langle c, b \rangle, \text{ and } \langle b, e \rangle \mapsto \langle b, e \rangle.
\]

Note that \( 1_c \) is an edge from \( c \) to \( c \) in the graph \( V(\text{Pa}(G_1)) \), which was written \( \text{Pa}(G_1) \) above. Let \( F = \hat{H} : \text{Pa}(G) \rightarrow \text{Pa}(G_1) \). Define \( \eta : P \Rightarrow P_1 \circ F \) as follows: \( \eta_a : \omega^2 \rightarrow \omega \) by \( \langle X, Y \rangle \mapsto Y \); \( \eta_b = \eta_c = \eta_d : \omega^3 \rightarrow \omega \) by \( \langle X, Y, Z \rangle \mapsto Y \); and \( \eta_e : \omega \rightarrow \{0\} \) is the constant 0. Each \( \eta_v \) is a (total) surjective function. The four commutativity conditions are trivially verified, as \( P_1 \) consists of the Y-components of \( P \) (but note that the edge \( \langle c, d \rangle \) doing no work is "collapsed" out of existence). Finally note that \( F : \text{Pa}(G) \rightarrow \text{Pa}(G_1) \) is surjective. In fact, \( \langle F, \eta \rangle \) is a projection. Therefore by Corollary 12, if \( P_1 \) terminates at \( e \) from \( a \), so does \( P \).
We now show that $P_1$ actually does terminate at $e$ from $a$. It should be noted that the level of detail given in the examples so far has an expositional purpose and exceeds that needed for the routine use of this method. With this example we become somewhat more terse. Let $P_2$ be the program indicated below, with underlying graph $G_1$ the same as for $P_1$, and with $\subseteq$ denoting the inclusion function. We take it as obvious that $P_2$ terminates, after exactly $n$ iterations around the loop.

Now define $\langle F, \eta \rangle : P_2 \rightarrow P_1$ by letting $F$ be the identity functor, and let each $\eta_v$ be an inclusion function. It is trivial to verify that $\langle F, \eta \rangle$ is a homomorphism, and in fact, a simulation. Therefore by Corollary 9, $P_2(a, e) \subseteq P_1(a, e)$, i.e., $\langle n, 0 \rangle \in P_1(a, e)$, for any $n \in \omega$. Therefore $P_1(a, e)$ is the constant function with value 0, and in particular, $P_1$ always terminates.

There is a rather interesting converse to Proposition 11.

**Proposition 13.** Let $\langle F, \eta \rangle : P_0 \rightarrow P_1$ be a program homomorphism such that for fixed $v, v' \in [P_0]$, $\eta_v$ is surjective and $\eta_{v'}$ is total. Then $P_1$ terminates at $Fv'$ from $Fv$ whenever $P_0$ terminates at $v'$ from $v$.

The main results of this section extend in various ways to nondeterministic programs and sets of nodes. For example, Proposition 13 applies exactly as it is to nondeterministic programs, and we also have the following.

**Proposition 14.** If $\langle F, \eta \rangle : P \rightarrow P_1$ is a program homomorphism such that for some $X \subseteq [P]$ and $v \in [P]$, $FG(v, X) = G_1(Fv, FX)$ (i.e., every path $Fv \rightarrow Fv'$ for $v' \in X$ is $Ff$ for some path $v \rightarrow x, x \in X$), and if $\eta_v$ is total, then $P_1$ terminates at $FX$ from $Fv$ implies $P$ terminates at $X$ from $v$.

**Proof.** By taking unions we have commutativity of

\[
P_v \xrightarrow{\eta_v} P_1 Fv \\
Q \xrightarrow{\eta_v} P_1 FX
\]
where \( Q = \bigcup \{ P(v, v') \circ \eta_{v'} \mid v' \in X \} \). Since \( \eta_{v} \) and \( P(Fv, FX) \) are total, so is \( Q \). But \( Q \subseteq \bigcup \{ P(v, v') \mid v' \in X \} \circ \bigcup \{ \eta_{v'} \mid v' \in X \} \), which is therefore also total. But this implies that \( \bigcup \{ P(v, v') \mid v' \in X \} = P(v, X) \) must be total. Q.E.D.

The above result also generalizes to nondeterministic programs.

7. UNFOLDMENTS

In this section we show how to “unfold” a graph, or a program, into its “best possible” loop-free form; i.e., into the “smallest” tree which covers it. The technique of unfolding is useful in exploring properties of programs, particularly in proving equivalence (see Section 8). This also bears some resemblance to the approach of Scott [17]. We begin with some preliminaries about graphs.

**Definition 8.** A graph \( G \) is loop-free iff for all \( v, v' \in \mid G \mid \), there is at most one path \( v \rightarrow v' \) in \( G \). A pointed graph \( G \) is \( \langle \mid G \mid, E, \delta_0, \delta_1, a \rangle \) such that \( \langle \mid G \mid, E, \delta_0, \delta_1 \rangle \) is a graph and \( a \in \mid G \mid \) is a vertex called the point of \( G \). A morphism \( \langle \mid G \mid, E, \delta_0, \delta_1, a \rangle \rightarrow \langle \mid G' \mid, E', \delta'_0, \delta'_1, a' \rangle \) of pointed graphs is a graph morphism \( \langle F, F \rangle : \langle \mid G \mid, E, \delta_0, \delta_1 \rangle \rightarrow \langle \mid G' \mid, E', \delta'_0, \delta'_1 \rangle \) such that \( \mid F \mid a = a' \). A pointed graph \( G \) is reachable iff for each vertex \( v \in \mid G \mid \), there is a path \( a \rightarrow v \) in \( G \). A tree is a reachable loop-free pointed graph. For reachable pointed graphs, and trees, call the point a root.

It is easily shown that a graph is loop-free in the above sense iff as an undirected graph it has no loops (the proof is omitted). Moreover, the above definition of tree is equivalent to other standard definitions of (possibly infinite) unordered (rooted) tree. We now give the basic construction for the unfoldment of a graph, and will later establish a number of its most important properties.

Let \( G \) be a pointed graph, with point \( a \in \mid G \mid \). Then the vertices of the unfoldment \( U(G) \) of \( G \) are the paths \( p : a \rightarrow v \) from \( a \) in \( G \), for all \( v \in \mid G \mid \); the edges of \( U(G) \) are the pairs \( \langle p, pe \rangle \) such that \( p \) and \( pe \) are paths from \( a \) in \( G \), and \( e \) is an edge of \( G \); \( \delta_0 : U(G) \rightarrow \mid U(G) \mid \) is defined by \( \langle p, pe \rangle \mapsto p \), and \( \delta_1 : U(G) \rightarrow \mid U(G) \mid \) is defined by \( \langle p, pe \rangle \mapsto pe \) (i.e., the \( \delta_i \) are projections); and the point of \( U(G) \) is the null path at \( a \), \( 1_a : a \rightarrow a \).

**Proposition 15.** \( U(G) \) is a tree.

**Proof.** Let \( p : a \rightarrow v \) be a node in \( U(G) \), say \( p = e_0 \ldots e_n \) with \( e_i \in G \). Then we claim that
\[
q = \langle 1_a, e_0 \rangle \langle e_0, e_0e_1 \rangle \langle e_0e_1, e_0e_1e_2 \rangle \cdots \langle e_0 \cdots e_{n-1}, e_0 \cdots e_n \rangle
\]
is a path from \( 1_a \) to \( p \) in \( U(G) \). Clearly \( \delta_0 q = 1_a \langle e_0 \rangle = e_0 \) and \( \delta_1 q = \langle e_0 \cdots e_{n-1}, e_0 \cdots e_n \rangle = e_0 \cdots e_{k+1}, e_0 \cdots e_{k+2} \rangle = e_0 \cdots e_{k+1} \) for \( 0 \leq k < n - 1 \). Thus \( U(G) \) is reachable.
We next show that for any $p \in |U(G)|$, except $p = 1_a$, there is exactly one edge in $U(G)$ with target $p$. Say $p = e_0 \cdots e_n$. Then any edge with target $p$ is of the form $\langle r, re_n \rangle$ with $re_n = p$. Therefore $r$ must equal $e_0 \cdots e_{n-1}$, and the unique edge is $\langle e_0 \cdots e_{n-1}, p \rangle$. It now follows that for any $p', p \in |U(G)|$, there is at most one path $p' \to p$ in $U(G)$. In fact, for $p' = e_0 \cdots e_n \neq 1_a$ there is only one edge to $p'$ in $U(G)$, so that if $p \neq p'$ the path $p' \to p$ must end with that edge, $\langle e_0 \cdots e_{n-1}, p \rangle$. Let $p_k = e_0 \cdots e_k$. Then if there is a path $p' \to p$, it must be the composite of a path $p' \to p_{n-1}$ with the edge $\langle p_{n-1}, p \rangle$. Now by backwards induction, we find the same for $p' \to p_{n-2}$, $\ldots$, so that in fact $p' = p_k$ for some $k$, and the path $p' \to p$ is of the form $\langle p_k, p_{k+1}, p_{k+2}, \ldots, p_{n-1}, p \rangle$.

If $p = p'$, the unique path $p \to p'$ is the null path at $p$. Thus $U(G)$ is loop-free, and is therefore a tree. Q.E.D.

$U(G)$ is a “best-possible” loop-free version of $G$ in the sense of satisfying a cofree property relative to $G$, much as $Pa(G)$ satisfies a free property relative to $G$, but dualized. For the path category, there was an inclusion $i_G : G \to Pa(G)$, injecting $G$ into $Pa(G)$ as a generating subject, and this inclusion was “universal” in the sense of there existing a unique morphism having certain properties with respect to any given other graph morphism $G \to V(C)$. In the present case we define a graph morphism $C_G : U(G) \to G$, expressing the covering or unfolding of loops in $G$ by linear paths in $U(G)$. We then show that $C_G$ has a certain universal property among all graph morphisms $T \to G$, where $T$ is a tree. The morphism $C_G$ is defined as follows: for $p \in |U(G)|$, $|C_G| p = \partial_1 p$; and for $\langle p, pe \rangle$ an edge in $U(G)$, $C_G\langle p, pe \rangle = e$.

**Fact 16.** $C_G : U(G) \to G$ is a pointed graph morphism.

We now prove the universal property.

**Theorem 17.** Let $G$ be a pointed graph, let $T$ be a tree, and let $F : T \to G$ be a pointed graph morphism. Then there is a unique pointed graph morphism $\tilde{F} : T \to U(G)$ such that $
abla$

$\tilde{F}$

commutes.

**Proof.** In this proof, write $C$ for $C_G$. Given $T$ and $F$, assume there is an $\tilde{F}$ such that $\tilde{C}\tilde{F} = F$. Then for any $v \in |T|$ we must have $|C| |\tilde{F}| v = |\tilde{F}| v$, i.e.,
\[ \partial \mid \vec{F} \mid v = |F| v, \text{ i.e., } |\vec{F}| v \text{ is a path in } G \text{ to } |F| v. \text{ For any edge } e : v \rightarrow v' \text{ in } G, \text{ we must have } C\vec{F}e = Fe; \text{ i.e., } \vec{F}e \text{ is an edge in } U(G) \text{ obtained by adding the edge } Fe \text{ of } G \text{ to the path } |\vec{F}| v \text{ in } G; \text{ i.e., } \vec{F}e \text{ is of the form } \langle |\vec{F}| v, (|\vec{F}| v)(\vec{F}e) \rangle. \]

Again let \( v \in \mid T \mid \). There is a unique path \( p : t \rightarrow v \) in \( T \), where \( t \) is the root of \( T \), because \( T \) is loop-free and reachable. Then \( Fp : Ft = a \rightarrow Fv \) is a path in \( G \), and \( \vec{F}p : Ft = 1_a \rightarrow \vec{F}v \) must be a path in \( U(G) \) such that \( C(\vec{F}p) = FP \). In fact, if \( p = e_0e_1 \ldots e_n \), then \( Fp = Fe_0Fe_1 \ldots Fe_n \), and \( \vec{F}p = Fe_0\vec{F}e_1 \ldots \vec{F}e_n \) must equal \( \langle 1_a, Fe_0 \rangle \langle Fe_0, Fe_0Fe_1 \rangle \ldots \langle Fe_0 \ldots Fe_{n-1}, Fe_0 \ldots Fe_n \rangle \), by the observations of the first paragraph. Since \( \partial \vec{F}p \) must equal \( |\vec{F}| \partial p \), it follows that \( |\vec{F}| v \) is the path \( Fp = Fe_0 \ldots Fe_n \).

Thus for \( e : v \rightarrow v' \) an edge in \( T \), \( \vec{F}e \) must be the edge \( \langle Fp, (Fp)e \rangle \) in \( U(G) \).

Conversely, if we do in fact define \( \vec{F} \) as was shown above to be necessary (namely, for \( v \in |G| \), \( |\vec{F}| v = Fp \), where \( p : t \rightarrow v \); and for \( e : v \rightarrow v' \), \( \vec{F}e = \langle (Fp), (Fp)e \rangle \)), then it must be verified that \( \vec{F} \) is a pointed graph morphism satisfying \( CF = F \). For the morphism part, let \( e : v \rightarrow v' \) in \( T \). Then \( \partial \vec{F}e = \partial \langle Fp, (Fp)e \rangle = Fe \), while \( |\vec{F}| \partial p = |F| p \) applied to the unique path \( f \rightarrow v \), i.e., \( |F| p \); and \( \partial \vec{F}e = \partial \langle Fp, (Fp)e \rangle = Fe \), while \( |\vec{F}| \partial e = |\vec{F}| v' \) applied to the unique path \( t \rightarrow v' \), i.e., \( F(p)e = (Fp)(Fe) \), since \( \partial \langle (Fp)Fe \rangle = \partial (Fe) = v' \). Moreover, \( |F| t \) is \( Fp \), where \( p \) is the unique path \( t \rightarrow T \), i.e., \( |F| t = F(1_a) = 1_a \), the point of \( U(G) \). Thus \( F \) is a pointed graph morphism.

We next show that \( CF = F \). First, \( |C| |\vec{F}| v = |F| v \). Let \( v \in \mid T \mid \). Then \( |C| |\vec{F}| v = |C| Fp = \partial (Fp) = |F| \partial p = |F| v, \) since \( p : t \rightarrow v \). Now let \( e : v \rightarrow v' \) in \( T \). Then \( (CF)e = C\langle Fp, (Fp)e \rangle = Fe \), as required. Thus \( \vec{F} \) satisfies the desired conditions.

Q.E.D.

In the language of a more esoteric category theory, the subcategory of trees is coreflective in the category of pointed graphs.

We now give an example of unfolding. Let \( G \) be the graph of Section 3. For simplicity of notation, let us use numbers to denote edges, 0 for \( \langle a, b \rangle \), 1 for \( \langle b, c \rangle \), 2 for \( \langle c, d \rangle \), 3 for \( \langle d, b \rangle \), and 4 for \( \langle b, e \rangle \). We take \( a \in \mid G \mid \) to be the point. Then \( G \) is reachable, but not loop free. The unfoldment \( U(G) \) has as nodes all paths \( p \) in \( G \) with \( \partial \partial p = a \). For a fixed \( v \in \mid G \mid \), the set of all paths \( a \rightarrow v \) in \( G \), though quite infinite (for \( v \neq 0, 1 \)), is easily described by a regular expression. The union of these gives a regular expression for all of \( \mid U(G) \mid \)

\[ 1_a \cup 0 \cdot (123)^* \cup (123)^* \cdot 1 \cup 0 \cdot (123)^* \cdot 12 \cup 0 \cdot (123)^* \cdot 4. \]

(For convenience we have omitted all set brackets, e.g., writing \{0\} as 0). The set of edges of \( U(G) \) is \( \{ (p, pe) \mid p, pe \in \mid U(G) \mid \text{ and } e \in G \} \), which is easily written explicitly as

\[ \{ (p, pe) \mid p = 1_a \text{ and } e = 0; \text{ or } p = 0 \cdot (123)^* \text{ and } e = 1; \text{ or } p = 0 \cdot (123)^* \cdot 1 \text{ and } e = 2; \text{ or } p = 0 \cdot (123)^* \cdot 12 \text{ and } e = 3; \text{ or } p = 0 \cdot (123)^* \text{ and } e = 4 \} \]
and can be drawn as

```
1
| 0 | 0 | 01 | 012 | 0123 | 01231 | 012312 | 0123123 | ... |
```

where there is no need to explicitly label the edges. We will give some more complex examples of unfoldments in the next section. Now a property of \( C \) which will be crucial for the applications to equivalence proofs.

**Proposition 18.** Let \( G \) be a pointed graph with point \( a \). Then for each path \( p : a \rightarrow v \) in \( G \) there is a (unique) path \( \tilde{p} \) in \( U(G) \) such that \( \tilde{C}_G \tilde{p} = p \). Moreover, if \( G \) is reachable, then \( \tilde{C}_G \) (the extension of \( C_G \) to paths) is surjective.

**Proof.** Say \( p = e_0 e_1 \cdots e_n : a \rightarrow v \) in \( G \). Then we claim that \( \tilde{p} = \langle 1_a, e_0 \rangle \langle e_0, e_1 \rangle \langle e_0 e_1, e_2 \rangle \cdots \langle e_0 \cdots e_{n-1}, e_n \rangle \) is a path in \( U(G) \) such that \( \tilde{C}_G \tilde{p} = p \). Clearly, \( \tilde{C}_G \tilde{p} = C_G \langle 1_a, e_0 \rangle \cdots C_G \langle e_0 \cdots e_{n-1}, e_n \rangle = e_0 \cdots e_n = p \). Uniqueness is proved as follows: let \( p_k = e_0 \cdots e_k \), let \( \tilde{p} = \tilde{e}_0 \cdots \tilde{e}_n \), let \( \tilde{p}_k = \tilde{e}_0 \cdots \tilde{e}_k \), and say that \( \tilde{e}_k = \langle q_k, q_k e_k \rangle \). Then \( C_G \tilde{p} = C_G e_0 \cdots C_G e_n = e_0 \cdots e_n \) implies that \( C_G \tilde{e}_k = e_k \), i.e., that \( \tilde{e}_k = \langle q_k, q_k e_k \rangle \). But \( q_k e_k = q_{k+1} \) since \( \tilde{e}_k = \tilde{e}_{k+1} \). Therefore \( q_n = q_{n-1} e_{n-1} = \cdots = q_0 e_1 \cdots e_n = e_0 \cdots e_n \) (for \( e_0 = \langle 1_a, e_0 \rangle \)).

Now assume \( G \) is reachable, and let \( f : v \rightarrow v' \) be a path in \( G \). Let \( p : a \rightarrow v \) be a path to \( v \) in \( G \) (this will not in general be unique). Then \( q = pf \) is a path to \( v' \) in \( G \). Let \( \tilde{p} \) and \( \tilde{q} \) be the paths in \( U(G) \) such that \( \tilde{C}_G \tilde{p} = p \) and \( \tilde{C}_G \tilde{q} = q \), and let \( \tilde{q}_0 \) be the initial segment of \( \tilde{q} \) of the same length as \( p \) and \( \tilde{p} \). From the above construction of \( \tilde{q} \) we know that \( \tilde{C}_G \tilde{q}_0 = p \). Therefore, if \( f \) denotes rest of \( \tilde{q} \) (to the right of \( \tilde{q}_0 \)), we have \( \tilde{q} = \tilde{q}_0 f \), and by functorality of \( \tilde{C}_G \) we have \( \tilde{C}_G \tilde{q} = \tilde{C}_G (\tilde{q}_0 f) = \tilde{C}_G (\tilde{q}_0) \tilde{C}_G (f) \), i.e., that \( q = pf \tilde{C}_G (f) \). Thus \( f \) is a path from \( p \) to \( q \) in \( U(G) \). Now by uniqueness of factorization of strings, \( q = pf \tilde{C}_G (f) = pf \tilde{C}_G (f) = q \). Q.E.D.

We now consider unfolding programs. This material is rather more difficult than the preceding, and can be skipped without serious loss of continuity, particularly the proof. Given a program \( P : G \rightarrow V(\text{Pfn}) \), we define \( U(P) \) by the composite graph morphism

\[
U(G) \xrightarrow{C_G} G \xrightarrow{P} V(\text{Pfn}),
\]

with underlying graph the unfoldment \( U(G) \) of \( G \). In particular then, for \( p \in \{ U(G) \} \), \(| U(P) | p = | P | C_G \{ p \} = P(\tilde{p}) \); and for \( \langle p, pe \rangle \) in \( U(G) \), \( U(P) \langle p, pe \rangle = P(\tilde{C}_G \langle p, pe \rangle) = P(\tilde{e}) : P(\tilde{\delta} e) \rightarrow P(\tilde{\delta} e) \). As with graphs, there is a "covering" morphism, \( C_P : U(P) \rightarrow P \), in this case a program homomorphism \( \langle C_G, \nu \rangle \),

\[
\text{FLOW DIAGRAM PROGRAMS}
\]

355
where \( \nu : U(P)^\prec \Rightarrow \hat{P}C_G \) (where \( \hat{C}_G \) denotes the extension of \( C_G : U(G) \rightarrow G \) to paths; but note that we often denote \( \hat{C}_G \) by just \( C_G \)) is defined, for \( p \in |Pa(U(G))| = |U(G)| \), by \( \nu_p = 1_{P(\partial_1 p)} : |U(P)^\prec| p = P(\partial_1 p) \rightarrow |\hat{P}C_G| p = |P| |C_G| p = P(\partial_1 p) \).

**Proposition 19.** \( C_p : U(P) \rightarrow P \) is a projection of programs.

**Proof.** According to Definition 5, we need \( \hat{C}_G \) to be surjective, we need each \( \nu_p \) (for \( p \in |U(P)| \)) to be an identity, and of course we need \( \nu : U(P)^\prec \Rightarrow \hat{P}C_G \). Proposition 18 shows \( \hat{C}_G \) is surjective. Each \( \nu_p \) is an identity by definition and thus it remains to show \( U(P)^\prec \Rightarrow \hat{P}C_G \). This follows from the definition \( U(P) = PC_G \) by direct application of Proposition 3. Q.E.D.

Again as with graphs, the covering morphism has a universal property. We first need some additional terminology about programs. A **pointed program** is a program \( \hat{P} : Pa(G) \rightarrow Pfn \) with underlying graph \( G \) having a point \( a \in |G| \) called the **entrance node** of \( P \). A **tree program** is one with underlying graph a tree. A **morphism of pointed programs** is a program homomorphism whose underlying graph morphism preserves the point. We need the notion of composition for program homomorphisms in order to state the universal property. Let \( \langle F, \eta \rangle : P \rightarrow P' \) and \( \langle F', \eta' \rangle : P' \rightarrow P'' \) be program homomorphisms. Then the composite \( \langle F', \eta' \rangle \circ \langle F, \eta \rangle : P \rightarrow P'' \) is the pair \( \langle F' \circ F, (\eta' \circ \eta) \rangle \), where \( \eta' \circ F : P' \circ F \Rightarrow (\hat{P}F') \circ F \) is the natural transformation defined by \( (\eta' \circ F)_v = \eta'_{F_v} \) for \( v \in |P| \), and \( (\eta' \circ F) \circ \eta : P \Rightarrow \hat{P}F'F \) is the composite defined by \( ((\eta' \circ F) \circ \eta)_v = (\eta' \circ F)_v \circ \eta_v \) for \( v \in |P| \). This notion of composition leads to a category \( Prog \) of programs. The following diagrams may conceivably help visualize the situations involved in the definition of composition.

![Diagram](image-url)

Composition of pointed program homomorphisms is defined exactly the same way, and one checks that the composite also preserves points.
THEOREM 20. Let $P$ be a pointed program, let $Q$ be a tree program, and let $\langle F, \eta \rangle : Q \to P$ be a pointed program homomorphism. Then there is a unique pointed program homomorphism $\langle \hat{F}, \hat{\eta} \rangle : Q \to U(P)$ such that

\[
\begin{array}{ccc}
Q & \xrightarrow{\langle F, \eta \rangle} & P \\
\downarrow \langle F, \eta \rangle & & \downarrow \hat{\eta} \\
U(P) & \xrightarrow{C_P} & P
\end{array}
\]

commutes.

Proof. Let the underlying graph of $Q$ be the tree $T$. Commutativity of the given diagram means that $C_G \hat{F} = F$ and $\hat{\eta} \circ (\hat{F} \ast \nu) = \eta$. But since $\nu$

\[
P_\nu(T) \xrightarrow{F} P_\nu(U(G)) \xrightarrow{\eta} Q \xrightarrow{\nu} P_\nu(G)
\]

is an identity morphism, this means that $\eta : \hat{Q} \Rightarrow \hat{P}F$ equals $\hat{\eta} : \hat{Q} \Rightarrow U(P)\hat{F} = \hat{P}C_G \hat{F} = \hat{P}F$. Now we already know by (Theorem 17) that the condition $C_G \hat{F} = F$ uniquely determines $\hat{F}$; and the above determines $\hat{\eta}_v = \eta_v : Q_v \to PF_v$ for $v \in |T|$. This gives uniqueness. For existence, we define $\hat{F}$ as in Theorem 17, and define $\hat{\eta} = \eta$. Then indeed $C_G \hat{F} = F$ and $\hat{\eta} \circ (\hat{F} \ast \nu) = \eta$.

Q.E.D.

The next section shows that $P$ and $U(P)$ actually have the same meaning, in an appropriate sense. This will provide a powerful technique for proving equivalence of programs. The fancy form of this result is that tree programs are a coreflective subcategory of pointed programs.

As a simple example, the unfoldment of the program $P$ of Section 3 is the earlier tree $U(G)$ of this section, with labeling by partial functions as follows:

\[
Z := 0 \quad Y > 0 \quad Z := 2Z + X \quad Y := Y - 1 \quad Y > 0 \quad Z := 2Z + X \quad Y := Y - 1 \quad Y > 0
\]

\[
\begin{array}{ccc}
& Y = 0 & \\
Y > 0 & \downarrow & Y > 0 \\
& Y = 0 & \downarrow
\end{array}
\]

for the label of the edge $\langle p, pe \rangle$ is $U(P)\langle p, pe \rangle = P(e)$, for $e \in G$.

8. EQUIVALENCE

From a general point of view, the determination of program semantics should perhaps be viewed as the reduction of a program to a particularly simple equivalent
form. For example, when we showed the program \( P \) of Section 3 computed the function \((2^y - 1) \cdot x\), we in effect reduced \( P \) to a very simple program using only familiar functions and having no explicit iterations. However, this is a little misleading, because in reality each of these familiar functions is defined by a (familiar) iterative program based on string manipulation of some radix representation. It is traditional to ignore this in the cases of multiplication and subtraction, since the time taken is proportional to the lengths (i.e., logarithms) of the numbers involved, and for small numbers may even be available almost instantaneously in hard-wired (hardware) form. For the function \( 2^y \) this idealization is much less realistic, as this function grows much faster in \( y \), takes much longer, and is unlikely to be hardwired (obvious algorithms involve loops with about \( y \), or better \( y^{1/2} \), iterations). With programs such as operating systems, which do very complicated things, one cannot expect genuinely simple descriptions of semantics. The most that would seem reasonable is a proof of the equivalence of an implemented program, which may be very complex but efficient, with an idealized program which serves as a semantic specification and should be relatively easy to understand but perhaps would be very inefficient if actually run.

For the practical problem of producing better programs, it would be desirable to have a good collection of transformations on programs, known to preserve equivalence, which (for example) increased efficiency (operating speed) or else simplicity (program size).

Thus, there are several very good reasons for wanting powerful methods for proving programs semantically equivalent. In this section we introduce a rather powerful notion of equivalence for programs, called "flow equivalence" and based on program homomorphisms. We show that flow equivalence implies semantic equivalence, and we show that a program and its unfoldment are flow equivalent. This leads to a general, useful, and possibly new, method for demonstrating the equivalence of programs, which we illustrate with an example somewhat less trivial than those of previous sections. The usefulness of this method stems from the fact that loop-free programs can well be easier to work with, even if they are infinite, than general programs.

**Definition 9.** A homomorphism \( <F, \eta>: P_0 \rightarrow P_1 \) is a flow equivalence morphism iff \( F \) is surjective and each \( \eta \) is a (total) identity function. Two programs are flow equivalent iff related in the equivalence relation generated by the pairs \( <P_0, P_1> \) such that there is a flow equivalence morphism \( P_0 \rightarrow P_1 \).

We use the modifier "flow" to distinguish this notion of equivalence from others in the literature. The following result helps to justify our terminology by showing flow equivalence implies semantic equivalence. (It is proved much like earlier results.)

**Proposition 21.** If \( <F, \eta>: P_0 \rightarrow P_1 \) is a flow equivalence and \( v, v' \in P_1 \), then

\[
P_0(F^{-1}v, F^{-1}v') = P_1(v, v').
\]
The above result holds just as well for nondeterministic programs and for any sets of entry and exit nodes. The following result also extends to nondeterministic programs.

**Proposition 22.** A program homomorphism is a flow equivalence iff it is both a simulation and a projection. In particular, for \( P \) a reachable program, \( C_\sigma : U(P) \to P \) is a flow equivalence.

**Proof.** If \( \langle F, \eta \rangle \) is both a simulation and a projection, then \( F \) is surjective and each \( \eta_v \) is a total surjective inclusion, i.e., each \( \eta_v \) is an identity, so that \( \langle F, \eta \rangle \) is a flow equivalence. The converse is evident. If \( G \) is reachable, Proposition 18 shows \( C_\sigma \) surjective, and by construction, each \( \eta_v \) is an equality. Q.E.D.

**Proposition 23.** If \( P \) is reachable program, then for any \( v, v' \in |P| \),
\[
U(P)(C_\sigma^{-1}v, C_\sigma^{-1}v') = P(v, v').
\]

**Proof.** By Proposition 22, \( CP : U(P) \to P \) is a flow equivalence, so Proposition 21 applies to give the desired equation. Q.E.D.

We now consider a less trivial example to illustrate our technique for proving equivalence. Kaplan [11], considering a problem posed by Paterson concerning two different flowchart representations of two-tape automata, sketches a proof of their equivalence requiring four pages of diagrams for at least eighteen steps in a certain formal theory of flowcharts. The two programs to be proved equivalent, hereafter denoted \( P \) and \( Q \), can be represented as below

\[
\begin{align*}
\text{where the nodes in the underlying graphs, denoted } G \text{ and } G', \text{ are as indicated, and are all labeled with the set } S \times S \text{ (with } S \text{ the set of "tape states"); where } f : S \to S \text{ is a total function (intuitively, what the machine does to the tape state in one read-compute-write step); and where } p, \bar{p} \text{ are complementary subfunctions of the identity}
\end{align*}
\]
1_S : S → S, i.e., \( p \cup \bar{p} = \phi \) and \( p \cup \bar{p} = 1_S : S → S \) (\( p \) might be "accept" and \( \bar{p} \) "reject"). Note we are using a special abbreviated notation for the functions on edges: \( fX \) is short for \( X : = fX \), i.e., the function \( \langle x, y \rangle \mapsto \langle fx, y \rangle \); \( pX \) is short for the partial subfunction of \( 1_{S \times S} : S \times S → S \times S \) defined on \( \langle x, y \rangle \in S \times S \) iff \( px \) is defined; similarly for \( fY \) and \( pY \); and then \( pX, pY \) is the subfunction of \( 1_{S \times S} \) defined if \( px \) and \( py \) are. For practical application of this technique to such small programs as \( P \) and \( Q \), unfoldments can be constructed directly from the pictorial representations of the graphs. For larger underlying graphs an automated procedure would be desirable.

To illustrate the way such procedure would work, we include a precise mathematical description of the construction of \( U(P) \) and \( U(Q) \); however, we use notation which is as convenient as possible for readers of this paper. Thus \( |G| = \{a, b, c, d, e\}, \ |G'| = \{a, b, c, d, e, f, g\}, \) and using a sequence of length two notation for edges, \( G = \{ab, ba, bc, cd, de\}, \) and \( G' = \{ab, ba, bf, bc, be, cd, de, fg, gf, ge\}. \) \( P(v) = S \times S \) for all \( v \in |G| \) and \( Q(v) = S \times S \) for all \( v \in |G'| \). For edges, we assume the pictures will enable the reader to produce the appropriate listings of partial functions.

To construct the unfoldment of \( P \), we first give regular expressions for the sets of paths from \( a \) to each node of \( G \) (these will be the nodes of \( U(G) \)). In alphabetical order of target node, these regular expressions are:

\[
(ab \cdot ba)^*, \ (ab \cdot ba)^* \cdot ab, \ (ab \cdot ba)^* \cdot ab \cdot bc \cdot (cd \cdot dc)^*, \\
(ab \cdot ba)^* \cdot ab \cdot bc \cdot (cd \cdot dc)^* \cdot dc, \ (ab \cdot ba)^* \cdot ab \cdot bc \cdot (cd \cdot dc)^* \cdot de.
\]

The edges of \( U(G) \) are the pairs \( \langle p, pe \rangle \) such that both \( p \) and \( pe \) are paths as described above, and \( e \) is an edge of \( G \). We refrain from giving an entirely explicit description (although it leads to a long expression, it is quite straightforward process, as in the example of Section 7). In order to determine the partial functions labeling the edges of \( U(P) \), we recall that \( U(P) \langle p, pe \rangle = P(e) \). Thus we arrive at a pictorial representation of the form
where again all nodes carry the set $S \times S$, and the partial functions on edges are indicated as for $P$. We have put special labels on the nodes in $C^{-1}_G(e)$: namely $(ab \cdot ba)^m \cdot ab \cdot bc \cdot (cd \cdot dc)^n \cdot de$ is denoted $<m + 1, n + 1>$.

For $Q$ (with underlying graph $G'$) we have in the same manner the following regular expressions:

$$(ab \cdot ba)^*, (ab \cdot ba)^* \cdot ab, (ab \cdot ba)^* \cdot ab \cdot bc \cdot (cd \cdot dc)^*,$$

$$(ab \cdot ba)^* \cdot ab \cdot bc \cdot (cd \cdot dc)^* \cdot cd,$$

$$[(ab \cdot ba)^* \cdot ab] \cdot [(bc \cdot (cd \cdot dc)^* \cdot cd \cdot de) \cup [be] \cup [bf \cdot (fg \cdot gf)^* \cdot fg \cdot ge]},$$

$$((ab \cdot ba)^* \cdot ab \cdot bf \cdot (fg \cdot gf)^*, (ab \cdot ba)^* \cdot ab \cdot bf \cdot (fg \cdot gf)^* \cdot fg,$$

yielding the following pictorial representation:

\[\text{Diagram}\]

in which we have once again used the notation $<m, n>$ to label nodes in $U(Q)$ covering the exit $e$ of $Q$. Here the correspondence is more complex; rewriting the expression for $G'(a, e)$, we see that its elements are of one of the following three forms:

$$(ab \cdot ba)^m \cdot ab \cdot bc \cdot (cd \cdot dc)^n \cdot cd \cdot de$$

$$(ab \cdot ba)^m \cdot ab \cdot be$$
Elements of the first form receive the label \( \langle m + n + 2, m + 1 \rangle \); those of the second \( \langle m + 1, m + 1 \rangle \); and those of the third \( \langle m + 1, m + n + 2 \rangle \). It is readily verified that the three sets of pairs of positive integers so obtained are disjoint, and among themselves exhaust all pairs of positive integers; the sets are \{\langle m, n \rangle | m > n\}, \{\langle m, n \rangle | m = n\}, and \{\langle m, n \rangle | m < n\}.

From Proposition 23, we conclude that

\[
P(a, e) = U(P)(1_a, F^{-1}e) \quad \text{and} \quad Q(a, e) = U(Q)(1_a, F^{-1}e).
\]

We are interested in proving that \( P(a, e) = Q(a, e) \). Thus, it will suffice to prove that \( U(P)(1_a, F^{-1}e) = U(Q)(1_a, F^{-1}e) \). Since \( F^{-1}e = \{\langle m, n \rangle | m, n > 0\} \) in both \( U(P) \) and \( U(Q) \), it will suffice to show, for each \( m, n > 0 \), that

\[
U(P)(1_a, \langle m, n \rangle) = U(Q)(1_a, \langle m, n \rangle).
\]

Here is where we really use the fact that unfoldments are trees: each \( U(P) \) and \( U(Q) \) have only and exactly one path \( 1_a \rightarrow \langle m, n \rangle \); let us denote the one in \( U(P) \) by \( \tilde{P}_{m,n} \) and that in \( U(Q) \) by \( q_{m,n} \). We have only to show that \( U(P)\tilde{P}_{m,n} = U(Q)q_{m,n} \); i.e., that \( P_{C_{\tilde{P}_{m,n}}} = Q_{C_{q_{m,n}}} \). But \( C_{\tilde{P}_{m,n}} = \delta_{1\tilde{P}_{m,n}} \) and \( C_{q_{m,n}} = \delta_{1q_{m,n}} \); let us denote these paths in \( G \) and \( G' \) by \( p_{m,n} \) and \( q_{m,n} \) respectively, noting they are also nodes in \( U(P) \) and \( U(Q) \) for which we have already worked out expressions. In fact,

\[
p_{m,n} = (ab \cdot ba)^{m-1} \cdot ab \cdot bc \cdot (cd \cdot de)^{n-1} \cdot cd \cdot de
\]

so that

\[
P(p_{m,n}) = (fX \circ \tilde{p}X)^{m-1} \circ fX \circ pX \circ (fY \circ \tilde{p}Y)^{n-1} \circ fY \circ pY,
\]

where for clarity we have written composition in the opposite-to-usual order (so that it corresponds to the concatenation expression). We consider \( q_{m,n} \) in three separate cases: if \( m > n \), we solve the simultaneous equations

\[
k + l + 2 = m, \quad k + 1 = n
\]

to obtain \( k = n - 1, l = m - n - 1 \), and

\[
q_{m,n} = (ab \cdot ba)^{k} \cdot ab \cdot bc \cdot (cd \cdot de)^{l} \cdot cd \cdot de
\]

so that

\[
Q(q_{m,n}) = (fX, fY \circ \tilde{p}X, \tilde{p}Y)^{n-1} \circ fX, fY \circ \tilde{p}X, pY \circ (fX \circ \tilde{p}X)^{m-n-1} \circ fX \circ pX;
\]

if \( m = n \), then \( q_{m,n} = (ab \cdot ba)^{m-1} \cdot ab \cdot be \), so that

\[
Q(q_{m,n}) = (fX, fY \circ \tilde{p}X, \tilde{p}Y)^{m-1} \circ fX, fY \circ pX, pY;
\]
finally, if $m < n$, we solve the equations

$$k + 1 = m, \quad k + l + 2 = n$$

to obtain $k = m - 1$, $l = n - m - 1$, and

$$q_{m,n} = (ab \cdot ba)^k \cdot ab \cdot bf \cdot (fg \cdot gf)^l \cdot fg \cdot ge$$

so that

$$Q(q_{m,n}) = (fX, fY \circ \bar{p}X, \bar{p}Y)^{m-1} \circ fX, fY \circ pX, \bar{p}Y \circ (fY \circ \bar{p}Y)^{n-m-1} \circ fY \circ pY.$$

We have now to check that $P(p_{m,n}) = Q(q_{m,n})$. In fact, it is easy to calculate that, in each case, we have

$$(fX \circ \bar{p}X)^m \circ pX, (fY \circ \bar{p}Y)^n \circ pY.$$  

Thus $P(a, e) = Q(a, e)$, as desired.

Several points should be made in connection with this example. First, the discussion just given constitutes a completely rigorous purely mathematical (in fact, purely algebraic) proof of semantic equivalence. We gave quite a few more details than would be necessary, or desirable, in a “handcrafted” or “blackboard” style proof (see [7] for an appropriate skeleton of the proof); the argument can be made quite clearly (though intuitively) from the pictures. However, we wanted to show the kinds of symbolic expressions which would be involved in a computer-aided attack on the problem. We also hope to have convincingly suggested that each step of the argument is either perfectly algorithmic (as obtaining regular expressions), or else is susceptible to straightforward–appearing heuristics. We hope to eventually have prepared a suitable interactive computer program to test the practical feasibility of these ideas.

The reader who wants to try for himself a simpler equivalence proof using the ideas of this paper might prove correctness of the program $P$ of Section 3 by showing it is equivalent to the program

$$\begin{align*}
Z &:= 2Z \\
Y &:= Y - 1 \\
Z &:= 1
\end{align*}$$

which incorporates an appropriate recursive definition of $2^y$.

Finally, it might be remarked that the techniques of this paper amount to reasonably compact and systematic ways of handling complex proofs by induction. It is well-known there is no algorithm for producing such proofs. Thus, it is entirely a matter of arranging things to be as transparent, and as amenable to heuristics, as possible. This
paper does not at all consider the problem of showing real numerical algorithms (i.e.,
those processing real numbers) equivalent in the traditional sense of numerical
analysis, which means asymptotically equivalent in some sense (e.g., for any $\epsilon > 0$,
if both programs are correctly initialized and run long enough, their outputs will
differ by less than $\epsilon$); see [9] for some discussion of this. It seems clear that arbitrarily
complex results from real analysis could be necessary as parts of such equivalence
proofs, and that while the results of the present paper would help with the logical, or
flow-of-control aspects, they are relatively independent of the purely analytic aspects.
On the other hand, the methods of this paper should be well suited to such essentially
exact problems as sorting and searching algorithms, and much of systems programming.

References

1. R. M. Burstall, An algebraic description of programs with assertions, verification, and
simulation, in “Proc. of an ACM Conf. on Proving Assertions about Programs,” Las
Cruces, New Mexico, 1972, pp. 7–14.
3. R. W. Floyd, Assigning meanings to programs, in “Proc. of Symp. in App. Math.,” Vol. 19,
5. J. A. Goguen, Systems and minimal realization, in “1971 IEEE Conf. on Decision and
6. J. A. Goguen, Mathematical representation of hierarchically organized systems, in “Global
7. J. A. Goguen, On homomorphisms, simulations, correctness, and subroutines for programs
and program schemes, in “Proc. of 1972 IEEE Symp. on Switching and Automata Theory,”
Computer Science and Category Theory, I: Basic Concepts and Examples,” IBM Watson
in “Proc. of an ACM Conf. on Proving Assertions about Programs,” pp. 66–73, Las Cruces,
New Mexico, 1972.
Report No. 12, Dept. of Computer Science, Univ. of Toronto, 1969.
on Semantics of Algorithmic Languages” (E. Engeler, Ed.), Lecture Notes in Math.,
14. R. Milner, “An Algebraic Definition of Simulation between Program,” A.I. Memo 142,
Preliminary draft, Courant Institute, N.Y.U., 1972.