SEMANTICS AND APPLICATIONS OF

FUNCTION GRAPHS

by

Robert M. Keller

UUCS-80-112

October 1980

SEMANTICS AND APPLICATIONS OF FUNCTION GRAPHS

Robert M. Keller

Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

28 October 1980

ABSTRACT

Function graphs provide graphical models of programs based on function application. The uses of such models include provision of a semantic framework for functional programs, explication of the structure of complex systems based on function application, increasing proximity of programs to certain application domains, resolution of ambiguities in programs based upon systems of equations, and representation of executable programs in machines based upon data flow execution. Application examples and underlying theory of function graphs are presented.

Keywords and Phrases: Applicative programming, Asynchronous systems, Distributed systems, Functional programming, Graph models, Concurrency, Data flow, Lambda calculus, Demand-driven computation, Parallelism, Multiprocessing

CR Categories: 3.2, 4.2, 5.2, 5.7, 8.1

This material is based upon work supported by the National Science Foundation under grant MCS 77-09369.

Table of Contents

List of Figures

1. Introduction

1.1. Motivating Function Graphs

1.1.1. Function graphs represent applicative programs and systems.

Function graphs are graphical representations of systems based on function application. We include programs within the scope of "systems". Programs based on function application are usually called "applicative", "functional", or "data flow" programs, although it may be seen that other types of programs can also be mathematically represented as function graphs.

1.1.2. Applicative languages simplify programming.

Some more important advantages of applicative programming languages include:

1. Greater system modularity
2. Ease in debugging
3. Natural exploitation of concurrency
4. Natural representation of communication
5. Ease in human comprehension

The features generally imply reduced programming costs. Additionally, since machines which directly execute applicative languages are being proposed ([Dennis and Misunas 74], [Plas 76], [Guzman and Segovia 76], [Arvind and Gostelow 77], [Gurd and Watson 77], [Rumbaugh 77], [Davis 78a], [Mudge 78], [Keller, Lindstrom, and Patil 79], [Cornish 79, 80], [Johnson, et al. 80], and others), the attendant reduction in number of software layers can further highlight the features listed above.

1.1.3. Graphical representations are often clearer.

As with other uses of graphs, function graphs are mathematically interchangeable with one-dimensional representations of the same system. However, they often serve to illustrate concepts more clearly and succinctly than their one-dimensional counterparts.

Furthermore, graphs can obviate the use of names for establishing relationships between entities, which is usually necessary when a one-dimensional representation is used for a basically graphical concept. For example, the lambda-calculus [Church 41], which has long been touted as a basic model for understanding of certain computational phenomena, requires a "re-naming" rule for its general application. With an appropriate graphical model, such as that presented herein, such rules are unnecessary.

As another example, the sequential listing of statements in a program text usually implies, or at least suggests, a corresponding sequentiality of control, much of which is inessential. A graph may used to illustrate only the essential sequentiality and, dually, the available concurrency. In contrast, many one-dimensional representations must be reprocessed to detect potentially concurrent operations (cf. [Keller 73, 75a, 75b]).

The present paper uses graphs to similarly expose other aspects of concurrent computation, such as the relationship between parallelism and choice of data types, and the comparison of programming styles for data-driven vs. demand-driven computation. It also uses graphs to demonstrate other aspects less related to concurrency, such as communication, binding, transformations of programs, and verification. We suggest that comprehensibility of such concepts may often be improved through the use of graphs.

1.1.4. Function graphs are a counterpart to flowcharts.

Function graphs bear a relationship to applicative programs similar to that of flowcharts to assignment-based programs. However, whereas flowcharts are usually thought to be informal representations of an algorithm yet to be coded and requiring further formalization prior to execution or analysis, function graphs rely only on the understanding of their constituent functions to be formally meaningful.

Like flowcharts, function graphs can be used informally to exhibit and develop basic interrelationships between system parts. However, this informal use can become a formal one if a programming language based on function graphs is available. In this case, there is no discontinuity involving the "coding" of the system specification in a given language, since the specification is already in a language which represents code. Thus the continuity provided by both developing a system specification and implementing it in terms of function graphs provides a more reliable design procedure than one involving a transition from specification domain to coding domain.

It is possible to enter function graphs directly into a programming system given a suitable graphical input device. Another possibility is to use a textual language which allows expression of function graphs in a manner in which the correspondence between the text and the graph is fairly direct. Indeed, combinations of textual and graphical entry can be profitable.

1.1.5. Function graphs form a basis for programming language semantics.

Additionally, function graphs may be used as a formal base for semantics of textual languages, even when the correspondence is not direct. The advantage of this form of base is that it provides a common means of program comparison and translation for different textual languages. In this sense, semantics of function graphs is related to other work on "denotational" semantics of programming languages (cf. [Stoy 77]).

A prototype language based on function graphs has been developed and is included as part of this presentation for sake of concreteness and illustration. While not all features of the language, called FGL (Function Graph Language), can be presented here, it is hope that the flavor of the graphical presentation and conceptualization can be conveyed.

1.1.6. FGL is both generic and specific.

FGL is related to a number of other language ideas which have appeared in the literature, in some cases in disjoint threads of investigation. As such, we hope that in addition to being a programming language, it provides a means of understanding these ideas, which include:

1. A large assortment of "data-flow" languages which are being proposed for other highly-concurrent machine architectures. Examples may be found in [Dennis 74], [Plas, et al. 76], [Arvind, et al. 77], [Davis 78b], [Weng 79].

2. Languages based on function application ([Church 41], [McCarthy, et al. 62], [Landin 64], [Kahn 74], [Burge 75], [Backus 78]) and on systems of equations ([Kleene 52], [Kahn 74], [O'Donnell 77]).

3. Data structuring operations from the Lisp family [McCarthy 60]. Although such operations fit naturally into FGL, we need not stop with just the conventional set of operations. The use of FGL graphs for representing data structures can replace the more machine-oriented "box diagrams" often found in texts dealing with data structuring. Further structuring ideas can also be found in [Keller 80b]

4. Languages which provide for programming with infinite structures, such as streams and trees ([Landin 64], [Kahn 74], [Burge 75], [Friedman and Wise 76]). Such structures are extremely powerful devices for modeling various mathematical structures and for representing communication among sub-systems. It has also been noted that they naturally provide many sites for the exploitation of concurrency in a highly-concurrent systems ([Friedman and Wise 78], [Keller, Lindstrom, and Patil 79]).

In summary, although we have a particular language FGL in mind, the reader may view FGL as if it represented a generic member of a family of languages based upon function graphs.

1.1.7. The following features further motivate the investigation of FGL:

1. FGL is an "applicative" language, in the sense that it is based on function application and therefore enjoys the features of such languages as expressed above.

2. FGL overcomes some of the awkward features of previous data-flow languages through a suggested implementation using "demand-driven" evaluation. FGL avoids the notion of "assignment". Rather than being called by the euphemism "single assignment" language (cf. [Tesler and Enea 68], [Chamberlin 71], [Arvind, et al. 77]) it is properly a "zero assignment" language.

3. FGL may be easily adapted to suit a large number of engineering and scientific applications. For example, signal-flow graphs used in digital signal processing (cf. [Rabiner and Rader 72]) and system dynamics models used for modeling and simulation of socioeconomic systems (cf. [Forrester 61]) can be naturally represented in FGL. It is attractive to have a general purpose, modularizable, language at hand to enhance such modeling approaches.

4. The graphical aspect of FGL has uses in software and hardware development by refinement. The use of graphical tools for software development has been mentioned before (cf. [Ross 77], [Weinberg 78], (Yourdon and Constantine 79], (Hebalkar and Zilles 79], and others). When similar tools are expressed in FGL, an additional advantage accrues: The graphs have a well-defined functional meaning, rather than simply representing procedure nesting, loop

nesting, calling sequences, etc. This meaning is a specification of the system under development.

5. The refinement of an FGL specification from a coarse interconnection of functions may proceed by further specifying those functions in the same language. There is no need to have different languages for "programming-in-the-large" vs. "programming-in-the-small" [DeRemer and Kron 76]. This is desirable, since often what initially appears to be a simple atomic task turns out to expand into something more formidable. Thus a transition made too early from a module interconnection language to a conventional programming language may result in substantial backing up in the design process. With the uniform language approach, when the level of atomic functions is reached, the specification is complete and the result is a runnable program.

6. FGL allows persons with little training to get started in programming. This is due to the few concepts involved and the absence of a need to acquire knowledge about a linear syntax. Given an adequate input device, a naive user need only know how to connect boxes and to interpret them as mathematical functions.

7. FGL allows the visualization of data structuring operations without using storage diagrams and references. As such, it exhibits the underlying concepts with a high degree of machine independence. It seems particularly useful in conceptualizing mathematically infinite data structures and representing the operations on such structures.

8. FGL is the base of directly-executable machine language, namely that for the system proposed in [Keller, Lindstrom, and Patil 79]. As such, it allows exploitation of concurrency without major concern from the programmer and narrows the gap between a high-level programming language and its machine implementation.

9. FGL allows intuitive representations of "functionals" or "higher-order functions" which are usually explained using the lambda calculus [Church 41]. We hope to show that FGL provides a better base for understanding the subtleties of these ideas.

10. FGL allows programs to be cleanly interfaced with file system files, in the spirit of [Balzer 71], [Ritchie and Thompson 75], [Friedman and Wise 77].

1.1.8. Function graphs may be executable or otherwise.

This paper presents ideas about function graphs on two levels. One level is that corresponding to executable programs. The other is a more general conceptual level, for which there may be no known efficient execution means. When it is necessary to contrast these levels, we shall refer to the former as special function graphs and the latter as general function graphs. The reasons for the desire to consider the general level at all are: .

1. Understanding the general level can often provide a clearer understanding of the special level.

2. Some ideas can be conceptualized only at the general level.

3. It is desirable to widen the special level as much as possible, i.e. to express more concepts in the form of executable programs. The general level provides a target for this widening.

1.2. How to and Why Read the Remainder of the Paper

The reasons for this paper are several:

1. To introduce the reader to graphical forms of applicative programming through a reasonably unifying model.

2. To provide a theoretical framework for those interested in such matters.

3. To survey the few key ideas present in an apparently importent, but embryonic, area of computer programming, including pointers to the literature for results which cannot be included here.

The section entitled Preliminery Discussion is intended to introduce some types of data objects and systems which can be explicated with graphical models and arguments. The section entitled Theoretical Basis may be read for those wanting a tutorial introduction to the theory behind such models. It may be skipped on first reading, or taken on faith. The section entitled Machine Evaluation of Computations Represented by Graphs further develops the function graph model, describes a simple language based on the model, and discusses a means of computing within the model. The section entitled Uses of the Graphical Formalism describes concepts which can be understood using the model, and manipulations and proofs within the model. Except for the sub-section on Loop Removal, this section may be skipped by those interested only in programming aspects. The Postlude mentions some of the historical aspects related to the ideas presented here, and summarizes the conclusions.

2. Preliminary Discussion

2.1. Computing with Infinite Objects and Equations

Several of the examples presented in this paper involve computing with infinite objects. By this, we mean that the program can manipulate as a whole objects which are conceptually infinite, even though the user may at any given run only wish to cause a finite truncation of the object to be manifest. In principle, the user could ask for the manifestation of an entire infinite object, whereupon if there were sufficient computing resources and he waited sufficiently long, any finite portion of the infinite object would be manifest. Since conventional theory of computation has shunned infinite objects, other than functions, in favor of working only with their finite truncations, a brief introduction to this style of computing is merited.

2.1.1. Infinite objects provide new ways of presenting algorithms.

There are several reasons for wanting to consider such objects:

1. Some systems, e.g. computer operating systems, treat their input and output (streams of requests and responses) as if they were infinite, since the point of termination of these streams is unknown and irrelevant.

2. Programming with infinite objects is often simpler than programming with finite objects, since it relieves the programmer of many concerns of "boundary conditions" which often are the cause of errors. For example, instead of writing a program to compute a finite set of values of a function

   $$f(1),\ f(2),\ f(3),\ \ldots..,\ f(n),$$

   the programmer might write a simpler program which computes the infinite set of values,

   $$f(1),\ f(2),\ f(3),\ \ldots$$

   and then use a pre-defined selection function to select the finite subset in which he is interested. Properly implemented, only the necessary values of f are really computed, but the programmer manipulates the series of values as though it were infinite.

3. With respect to this paper, one of the prime uses of function graphs is to display program structures which represent an efficient and applicative method for computing such infinite data

structures.

2.1.2. An example of computing with infinite objects:

Let us give a simple example of defining an infinite structure. Suppose we wished to define the (infinite) sequence of all odd natural numbers,

1 3 5 7 ...

(Here, and throughout the paper, three dots indicates a sequence which continues *ad infinitum*, whereas four dots indicates a sequence with a last component.) We may do so by producing a general function odd_from which with argument n produces

n n+2 n+4 ...

then applying that function to argument 1. To define odd_from, we simply note that it satisfies the equation

odd_from(n) = n followed_by odd_from(n+2)

where followed_by is a binary function which produces a sequence consisting of the item on its left followed by the item on its right. Our sequence is then given by the result of odd_from(1).

To spell out in detail,

```
odd_from(1)
    = 1 followed_by odd_from(3)
    = 1 followed_by 3 followed_by odd_from(5)
    = 1 followed_by 3 followed_by 5 followed_by odd_from(7)
    = ...
```

For readability and convenience, we henceforth omit the followed_by in writing such sequences, preferring to write

1 3 5 ...

instead of the last line above.

We also use the expression cons(x, y) in place of x followed_by y, since a minor extension of the cons (constructor) function from Lisp is just what we need to implement followed_by.

2.1.3. More examples involving infinite objects.

To give a further example of operating on infinite objects, we know that the sum of the first n odd numbers is equal to the square of n. We could therefore compute the stream of squares by a function which produced successively the sum of the components of its input stream. Let us call such a function sum_stream. The first component of sum_stream(x) is just the first component of x. Using head(x) to refer to this component, we see the basic form

```
sum_stream(x) = cons(head(x), ....)
```

where the dots must yet be filled in to give us a complete definition.

We now observe that if we knew sum_stream(x), then we could add its components pair-wise to the tail of the input (i.e. the components which are followed by the head) and end up with exactly sum_stream(x).[1] In other words, we have an equation

```
sum_stream(x) = cons(head(x), add_streams(tail(x), sum_stream(x)))
```

Here we have used add_streams to name the function which adds two streams component-wise. For example,

```
add_streams(3 5 7 9...., 1 4 9 16...) = 4 9 16 25...
```

To see that equation for sum_stream gives us exactly the information needed, we try to discover what it tells us about sum_stream(1 3 5 ...). Using the definition,

```
sum_stream(1 3 5 ...) =

cons(1, add_streams(3 5 ...., sum_stream(1 3 5 ...))) =
```

[1] We avoid using the Lisp car and cdr for head and tail for two reasons: One is that these terms are not suggestive of their meaning, and the other is that we have in mind a later extension of cons for which head and tail fit more nicely.

cons(1, add_streams(3 5 ..., cons(1, add_streams(3 ..., sum_stream(1 ...)))))

= cons(1, cons(4, cons(9, ...)))

While this type of reasoning may appear foreign to the reader at first, with a little practice, it is easy to become convinced that It is an extremely powerful definitional and programming tool.

Incidentally, we could go further and provide a definition of add_streams: add_streams(x, y) = cons(head(x)+head(y), add_streams(tail(x), tail(y))), where + represents the usual addition operator on two numbers.

2.2. Graphical Models

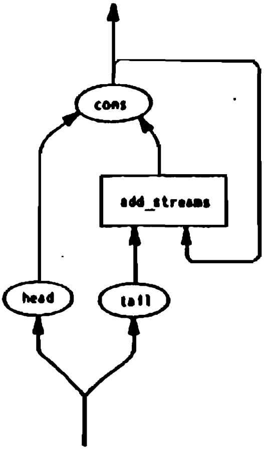


Figure 2-1: A graph for Function sum_stream

2.2.1. Graphical models clarify complex ideas.

We are interested in graphical expression of computational specifications of the type specified in the previous section. For example, the function sum_stream could be represented by the graph shown in Figure 2-1. This graph illustrates the input and output of the function, as well as the essentials of its internal structure.

2.2.2. Arc names are irrelevant in function graphs.

In most cases we shall avoid giving names to the arcs of a graph. Instead, we shall rely on the orientation of the arcs to determine the position of the corresponding argument to a function. That is, viewing the node so that input arcs enter a node at its bottom, the left-to-right orientation of arcs corresponds to the order of argument listing. Where ambiguity or confusion might arise, we can return to the use of naming.

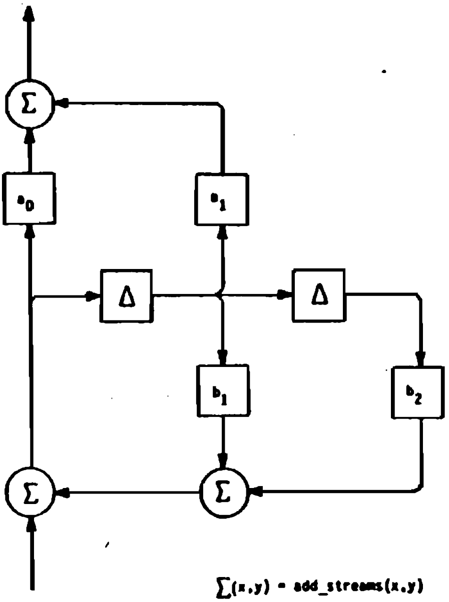
2.2.3. Fields outside Computer Science employ formalizable graphs.

It is claimed that for certain examples, the graphical expression can lend to better comprehension of structures. Similar representations have occurred in related application areas. For example, in digital signal processing, digital filters are often represented graphically. Unfortunately, the behavior of such filters is often explained using notions such as clocks, unit time delays, and other jargon, instead of appealing to their intrinsic meaning in terms of functions on sequences. The legend accompanying Figure 2-2 illustrates how the basic filter operations can be viewed as functions on streams defined in the previous section. By making this connection, it is possible for Computer Science to contribute to digital filter design by:

1. Providing a language in which the ideas of filtering can be expressed directly, instead of having to revert to Fortran coding, which is often a lengthy process.

2. Allowing the digital filter researcher to embed his filters directly into a general purpose computational system (i.e. a Function Graph Language).

The language Lucid [Ashcroft and Wadge 77] is a textual one based entirely upon recurrence equations of the type used in defining stream functions as



$\sum(x,y)$ = add_streams(x,y)

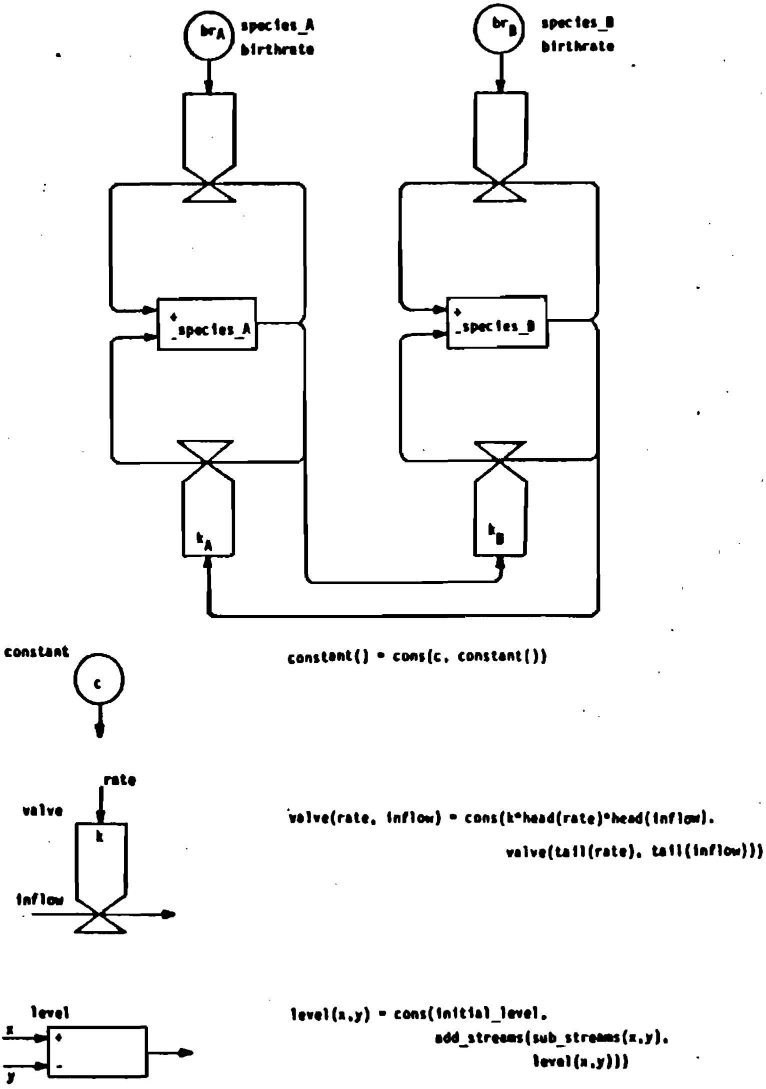
$\Delta(x)$ = cons(0,x)

$a_i(x)$ = cons(a_i*head(x), a_i(tail(x)))

Figure 2-2: Graphical representation of a digital filter having a transfer function with z-transform $(a_0+a_1 z^{-1})/(1-b_1 z^{-1}-b_{2z}-2)$

used above.

Another application area of interest is that of <u>System</u> <u>Simulation</u>. Here one is often concerned with physical processes which can be modeled as intercommunicating via streams of discrete values. An example is Forrester's <u>system</u> <u>dynamics</u> [Forrester 61], which employs graphs of the type shown in



```
constant() = cons(c, constant())

valve(rate, inflow) = cons(k*head(rate)*head(inflow),
                           valve(tail(rate), tail(inflow)))

level(x,y) = cons(initial_level,
                  add_streams(sub_streams(x,y),
                              level(x,y)))
```

Figure 2-3: Graphical representation of a system dynamics model

Figure 2-3. As with digital filters, the nodes in this graph can be defined as stream functions, as presented in the legend. A language, Dynamo [Pugh 70], already exists which allows such models to be input to computers. Although Dynamo is indeed an applicative language (perhaps the first such), its textual expression rather resembles a monolithic Fortran program, which does little to help its user visualize relationships between sub-systems. By using a textual language (as we describe later) which is isomorphic to the graphical model, one attains a new degree of modularity, at the same time retaining the possibility of having one's models embedded in a general purpose computational system. Some other simulation methods, e.g. [Pritsker and Pegden 79], employ a graphical notation, but these graphs do not have the formal properties of functionality in which we are interested.

Other isolated instances of function graphs have occurred from within computer science, such as in [Smith and Chang 75], who employ graph transformations to illustrate query optimizations for relational databases.

2.2.4. Stream components need not be simple.

One should not infer from the above examples that FGL deals only with streams of atomic (i.e. indecomposable) values. The components of a stream might well be arbitrary structures (including possibly streams) themselves. For example, in Figure 2-4 an argument tree becomes the first of a stream of trees, the rest of which is obtained by splitting its non-atomic members into sub-trees. This example exhibits an cyclic arrangement of "processes" which communicate via streams and perform appropriate stream functions such as filtering of atoms or non-atoms. The figure shows the basic communication scheme, but the functions inside the boxes may be described by simple conditional expressions corresponding to acyclic graphs:

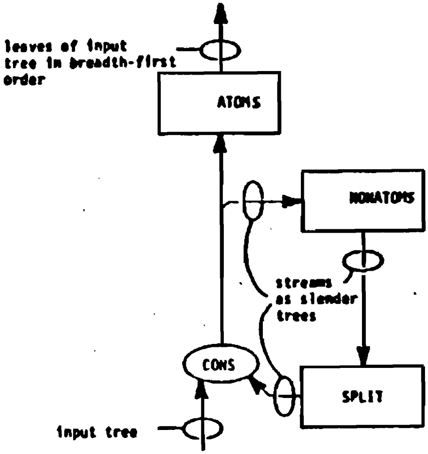


Figure 2-4: A function which strips the leaves from a tree in breadth-first order.

```
atoms(x) =
        if null(x)
                then nil()
                else if atom(head(x))
                        then cons(head(x), atoms(tail(x)))
                        else atoms(tail(x))

nonatoms(x) =
        if null(x)
                then nil()
                else if atom(head(x))
                        then nonatoms(tail(x))
                        else cons(head(x), nonatoms(tail(x)))

split(x) =
    if null(x)
        then nil()
        else cons(head(head(x)), cons(tail(head(x)), split(tail(x))))
```

In the above example, nil is a function which produces a terminated empty stream, whereas null is a predicate which tests whether its argument is that stream. atom tests whether its argument is an atom. head and tail, acting on trees which are not atoms, extract the left and right subtrees, respectively.

2.3. Semantics of Function Graphs

2.3.1. Function graphs have a simple basic form.

The main interest here is in systems of computation which can be represented as a certain form of *directed graph*, which we are calling a "function graph". The name derives from the interpretation of the nodes of the graph as functions. The arcs of a function graph represent variables ranging over *data structures* (including various degenerate forms of this concept). Arcs directed from one node to another therefore represent the phenomenon of the first node creating a data structure which is an input to the second.

It is possible for an arc to "fan-out", i.e. split into two or more arcs, indicating that the *same* structure is to be made available to more than one node as input, as shown in Figure 2-5. No meaning is assigned to two or more arcs converging together. Other than this restriction, any interconnection of nodes can be ascribed a meaning, as shall be seen.

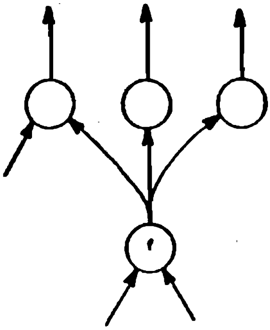


Figure 2-5: Illustration of fanout

The following features will be seen to fit into this general model:

(i) Creating data structures by several functions concurrently.

(ii) Operating upon data structures at the same time that they are being created.

(iii) Representation of communication protocols.

(iv) Representation of infinite graphs by finite means.

(v) Representing "history-dependent" functions.

(vi) Resolving ambiguity in the representation of recursively-defined functions.

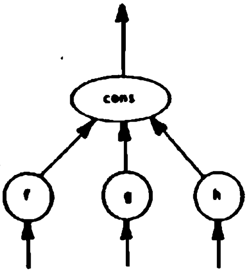


Figure 2-6: Concurrent creation of data structures by f, g, and h

Item (i) suggests that such a data structure should somehow be represented as several function nodes sharing a single output arc. Although this possibility was excluded above, we can represent this sharing by including another node, the output of which is the data structure and the input arcs are directed from several function nodes, as in Figure 2-6.

Concerning item (ii), our formalism does not require that the entire data structure at the output of a function node be completely present at any instant. Instead, the structure appears <u>during</u> the computation, possibly a piece at a time. It is even proper, and often convenient, that we consider computations of infinite duration which produce data structures of infinite extent.

Regarding item (iii), at some levels of detail, familiar notions of communication protocol may be completely abstracted from view. However, when such issues are of concern, they may often be represented in our formalism.

Regarding item (iv), the three primary techniques for representing infinite objects, either data structures or function graphs, are the use of cycles in graphs, the use of graph productions, and allowing data objects which can themselves be function graphs. These techniques shall be explained and interrelated in the subsequent development.

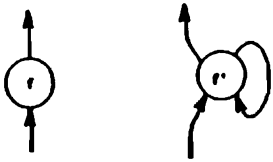


Figure 2-7: Recoding a history dependent function

Regarding item (v), there is really no need to introduce a special notion of a "history-dependent" function, since our model allows the encoding of an arbitrary history as a data structure. Any function with a form of state may be represented by a node with a self loop which feeds the previous history of the function back to itself at each computational step, as depicted in Figure 2-7.

Finally, regarding (vi), we shall see in subsequent sections that there are two ways of interpreting an equation such as that which defines sum_stream. The choice of interpretation has bearing on storage and execution efficiency, so it will be useful to resort to a graphical representation of the function, which resolves the ambiguity.

2.4. Representing Systems as Graphs

Given that one accepts the basic premises of function graphs as presented thus far, we now wish to further stipulate the nature of node functions and arc data structures. For this purposes we shall use streams as our data structures, although the basic ideas will later be seen to generalize. Begin by imagining that we observe the output of a function node over a semi-infinite computation period, that is, one which has a definite start but no finish.

2.4.1. The null structure contains no information.

Assume that the data structure starts out initially as a special _null structure_, which we denote

$$?$$

After some elapsed time, the node function produces some output, changing the structure to s_1. After more elapsed time, it gets changed to s_2, then s_3, and so on. Over the observed period we therefore see

$$?,\ s_1,\ s_2,\ s_3,\ \ldots.$$

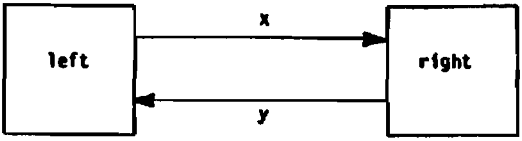


Figure 2-8: Handshaking example

2.4.2. Handshaking illustrates a simple form of communication.

Consider two devices communicating via a simple "handshaking" protocol, as in Figure 2-8. Assume that the left node initiates communication by sending a signal b on the top line. When the signal is received by the right node, the latter responds by sending a signal c on the bottom line. When the left node receives this signal, the whole process starts over again.

If we record the accumulated signals in a string on each line in a state, we get the following state-transition picture:

```
    ?         b         b         bb        bb         bbb
       -->       -->       -->       -->        -->         -->   ...
    ?         ?         c         c         cc         cc
```

One way of expressing the above behavior is to give a set of productions which characterizes the transitions between states. From an understanding of this behavior, the following productions suffice:

```
              x       bx
                -->              if length(x) = length(y)
              y       y


              x       x
                -->              if length(x) ≠ length(y)
              y       cy
```

Here x and y represent arbitrary finite strings and length(x) is the length of x.

A second way of representing the behavior is to present the left and right nodes as functions, in the form

$$\text{left}(y) = \text{cons}(b, \text{invert}(y))$$

$$\text{right}(x) = \text{invert}(x)$$

where invert(x) is the string obtained by replacing each b in x with c and each c in x with b.

Taken individually, the functions defined do not capture the short term hand-shaking behavior of the system. However, taken together, with the understanding that the system when in state (x,y) tends toward the state (left(y), right(x)), they do quite well. For example,

$$\begin{matrix} \text{left}(?) & & b \\ & = & \\ \text{right}(?) & & ? \end{matrix}$$

$$\begin{matrix} \text{left}(?) & & b \\ & = & \\ \text{right}(b) & & c \end{matrix}$$

$$\begin{matrix} \text{left}(c) & & bb \\ & = & \\ \text{right}(b) & & c \end{matrix}$$

$$\begin{matrix} \text{left}(c) & & bb \\ & = & \\ \text{right}(bb) & & cc \end{matrix}$$

and so forth.

2.4.3. Solving equations expresses long-range system behavior.

The functional description is able to express one aspect of the system succinctly which the state-transition behavior cannot, namely that there will be no deadlock in the sense that some node eventually stops sending signals to the other. To see this, we first submit that the long-range behavior of the system is a solution (or fixed-point) (x, y) of the system of equations

$$x = \text{left}(y)$$

$$y = \text{right}(x)$$

From the discussion regarding state-transition behavior, it is intuitive that the solution should be

x = bbb...

y = ccc...

Indeed, this is a solution, since

left(ccc...) = b invert(ccc...) = bbbb...;

right(bbb...) = invert(bbb...) = ccc...

We have not demonstrated that the above solution is unique, nor how that is the proper choice among several possibilities. This will be addressed in the following sections.

2.5. Recursion

It is useful to extend our concept of graphs to graphs which are specified by *graph grammars*. This extension allows us to represent infinite graphs by finite presentations, which will give us a convenient means of defining functions by possibly recursive applications of *productions*.

Suppose that we allow the nodes of a graph to be labelled with two types of symbols: *terminal symbols*, which denote pre-defined functions, and *auxiliary symbols*. For each auxiliary symbol, there is to be exactly one *production* which has the node labelled with the auxiliary symbol as *antecedent*, and an accompanying graph as the *consequent*. The set of productions collectively will sometimes be called a *graph grammar*.

Nodes labelled with terminal and auxiliary symbols will be called *terminal nodes* and *auxiliary nodes* respectively. We assume a one-to-one correspondence between the arcs of any node labelled with an auxiliary symbol and unconnected arcs in the consequent of the production. The meaning ascribed to a production is that whenever there is an auxiliary node in the graph, it may be replaced with the consequent of its corresponding production to determine its meaning.

For enhanced readability, we shall adopt the practice of making nodes containing terminal symbols circuler or elliptical, and nodes containing auxiliary symbols rectangular.

Furthermore, we shall use hexagonal nodes to symbolize an arbitrary subgraph, such as the consequent of a production.

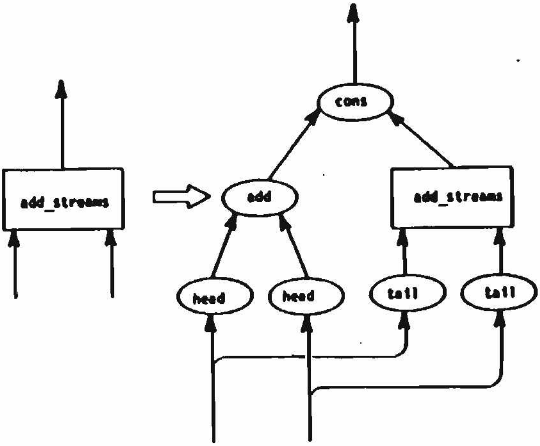


Figure 2-9: A production for the add_streams function

2.5.1. Example:

Consider the add_streams function used earlier. We can represent this function in terms of a more primitive function add which adds only a single pair of integers, using the production in Figure 2-9.

As further examples of recursion, we show below two different examples, both of which generate all odd prime numbers. The modus operandi of these two examples is suggested in Figures 2-11 and 2-13. The detailed definitions of some of the operators contained therein are presented in the Evaluation section which appears later.

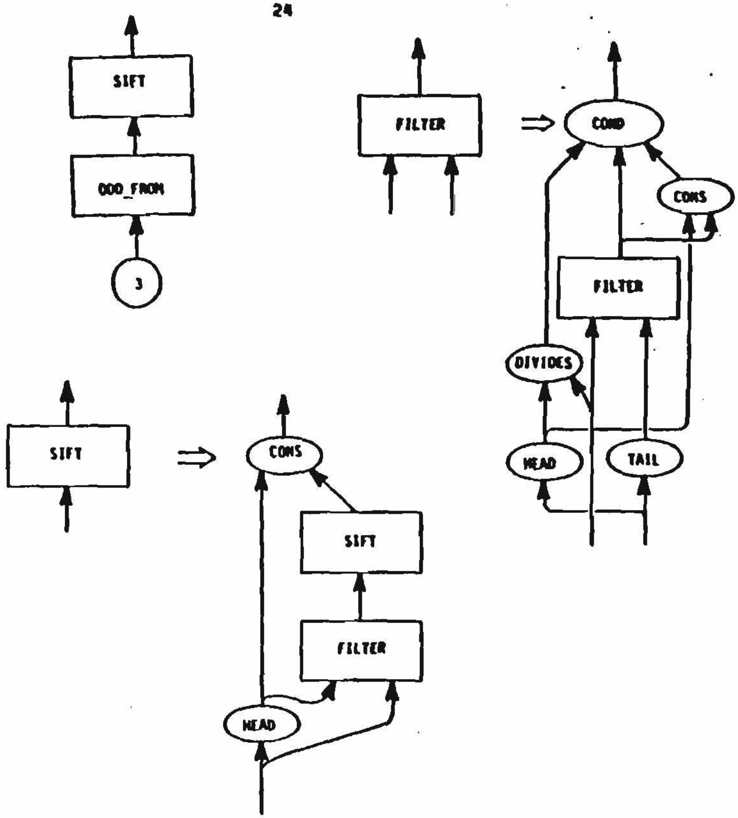


Figure 2-10: First Odd-Primes Example

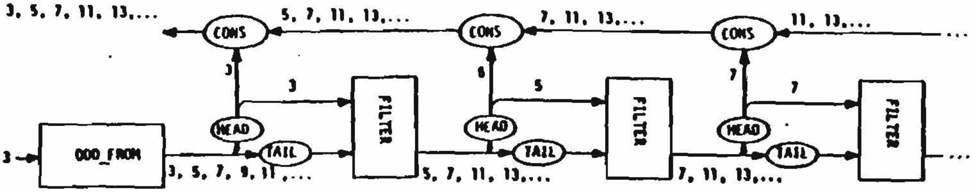


Figure 2-11: Expansion of the First Odd-primes Example (odd_from is defined in Section 2.1.2).

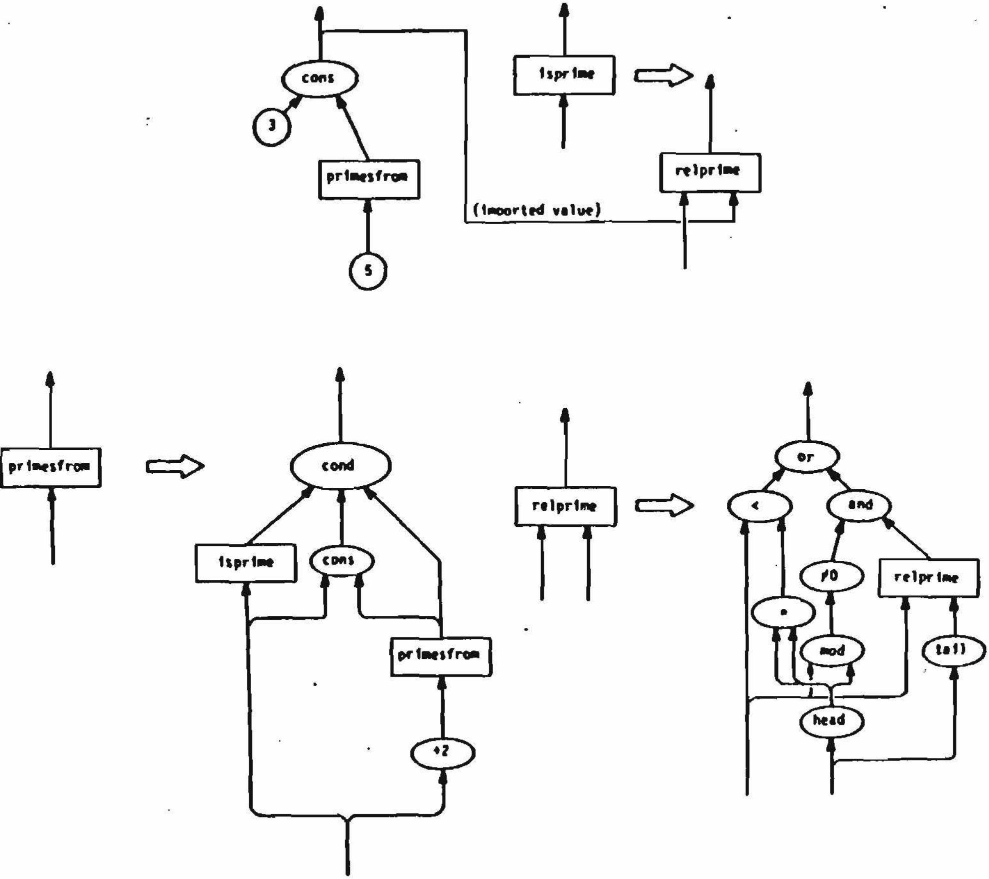


Figure 2-12: Second Odd-Primes Example

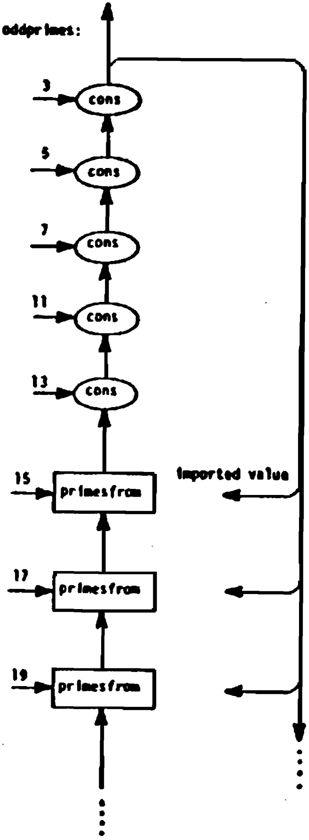


Figure 2-13: Expansion of the Second Odd-primes Example

2.6. Splitting Transformation

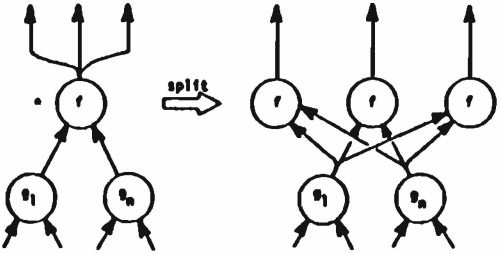


Figure 2-14: Splitting transformation

The discussion of recursion in the previous section described ways of transforming a graph by applying productions. Another type of transformation of interest involves local modifications to the graph based on the fact that the nodes represent functions. Such transformations are useful in understanding the functions represented by graphs. However, these transformations are not necessary to provide meaning for the graphs. That can be done on a purely functional basis, as described in the section on Theoretical Basis.

Because nodes of a graph represent functions, it is easy to see the validity of the splitting transformation, as demonstrated in the diagram of Figure 2-14. Because the values on the top arcs each represent $f(x_1,....,x_n)$, where each $x_i = g_i(....)$, we can split f into several copies of itself in place of the split output arc of f. One reason for wanting to do this might be that we wish to make further transformations involving just one of the copies of f. The splitting transformation exemplifies the notion of referential transparency (cf. [Quine 60], [Landin 64]), in that a functional expression has the same meaning independent of its context.

Notice that to say that the splitting rule is applied does not remove the ambiguity of where in the graph it is applied. We shall adopt the practice of placing an asterisk near the node being split to so indicate.

Applying the splitting transformation to the example in Figure 2-1, we get the sequence shown in Figure 2-15. The infinite graph is again seen to be embedded in the limit of an infinite succession of such transformations.

It can be noted in Figure 2-15 how function graphs subsume the usual "box diagrams" (cf. [Allen 78]) used to represent data structures in Lisp-like languages. The cons nodes replace the role of boxes containing "dotted pairs". The arrows are reversed in going from one representation to the other, in the sense that they represent references in box diagrams, but data flow in function graphs. Furthermore, in function graphs, such data structure nodes blend well with functions other than cons, whereas no blending suggests itself with box diagrams.

The inverse of splitting, which will be called folding, will also have its uses in discovering certain equivalences later on.

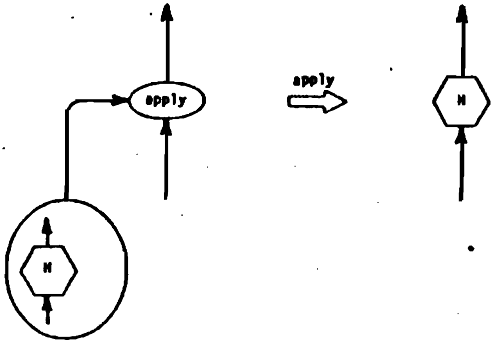


Figure 2-16: The meaning of apply

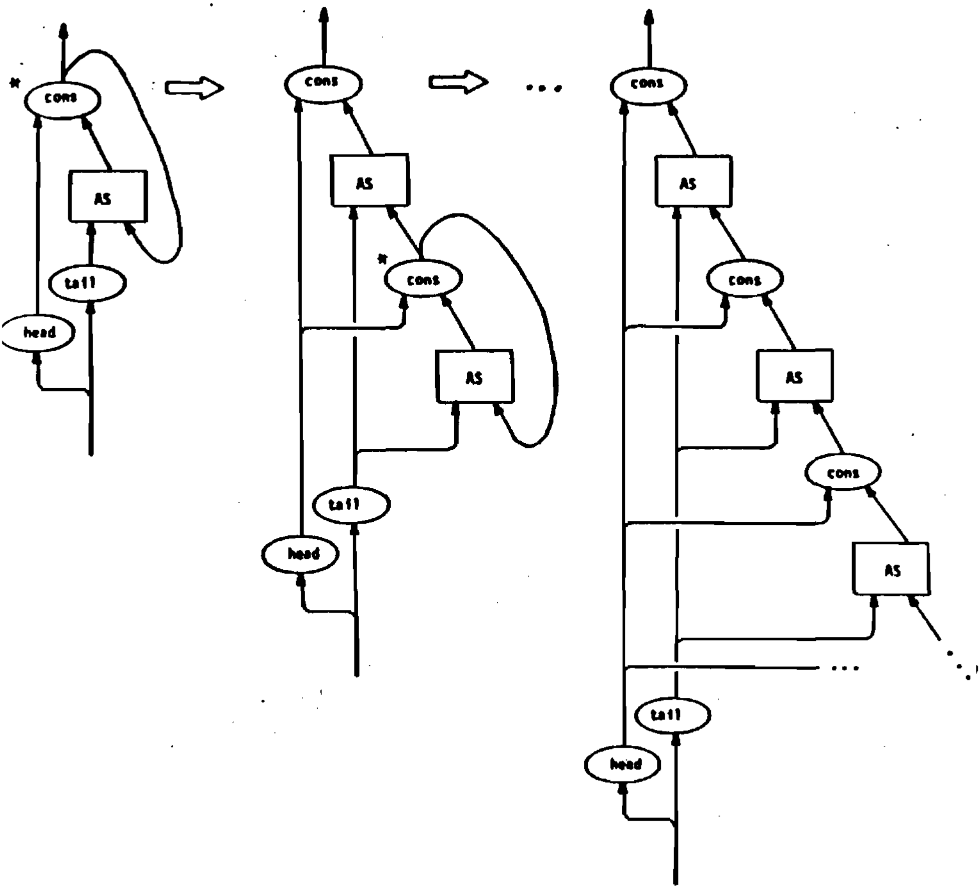


Figure 2-15: Repeated application of the splitting transformation

2.7. Funotions as Values

In order to present a semantics of productions, and also to represent a powerful definitional mechanism in graphical terms, we introduce a special function called apply. In its simplest form, apply is a function of two arguments, one of which is a function and the other of which is an argument to that function. Functions which take one or more functions as arguments are sometimes called "functionals". (Analogously, the corresponding graphs might be called "graphicals".)

2.7.1. Enveloping shows the creation of function values.

We might assume that there are some primitive function objects which can be used as the first argument to apply. However, it is also desirable to be able to create function objects ourselves.

In FGL, a function which can be used as an argument to another will be shown by enveloping the former inside a node of a graph. That is, the envelope is a constant function which produces the enveloped function of its value. The meaning of apply can then be expressed by the rule shown in Figure 2-16. It is not difficult to see that if the domain of the first argument to apply is D_1 --> D_2, the set of functions from D_1 into D_2, and the domain of the second argument is D_1, then apply is a function from (D_1 --> D_2) x D_1 into D_2.

2.7.2. Import arcs provide extra flexibility.

It is important that we allow import arcs to pass from the outside to the inside of an envelope. This allows the graph inside of the envelope to get values from the outside in one of two ways:

1. By means of arguments which are bound to the free input arcs inside the envelope when the latter is applied.

2. By means of import arcs which pass into the envelope. These arcs are present either in the initial graph, or residual from prior applications.

Function values which have their import arcs connected to the outside world are often called closures [Landin 64]. As an example, suppose that we wish to define a function serial_comp of two arguments, each of which is a function

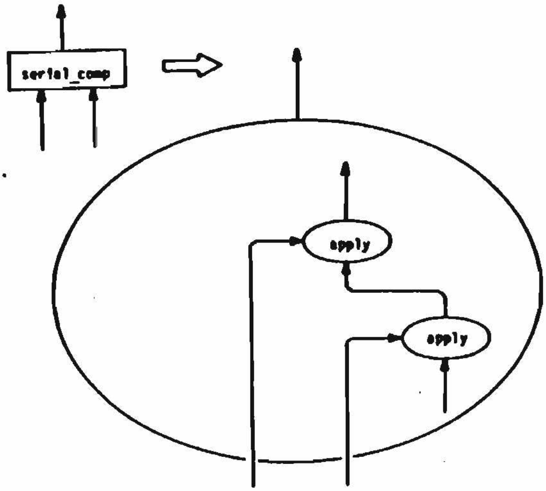


Figure 2-17: Illustration of function envelopes

itself, such that the result of serial_comp(f, g) is a function, say h, such that h(x) = f(g(x)). In other words, h is the serial composition of functions f and g. A graphical presentation of serial_comp is shown in Figure 2-17. The envelope shown as the consequent of the production for serial_comp has the functions f and g as imports. When this envelope is presented to the apply operator, the envelope is stripped off and the free input arc inside is bound to the second argument of the apply. Functions such as serial_comp which are designed to take functions as arguments are sometimes called "combinators".

For sake of further illustration, we direct the reader to Figure 2-18, which illustrates the concept commonly called "Currying". Here a binary function is represented as unary function, the value of which is another unary function. Applying the binary function to (x, y) is the same as applying the unary

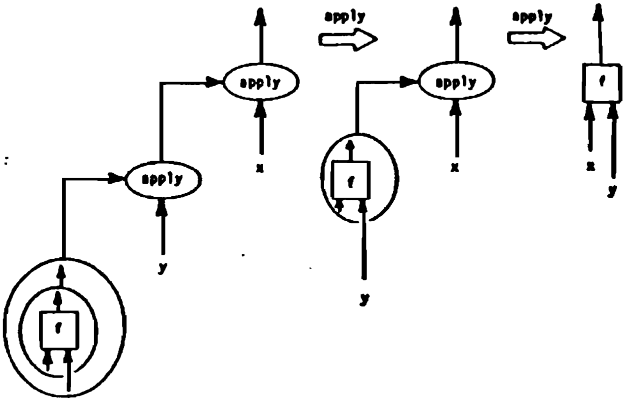


Figure 2-18: Illustration of Currying

function to y, then applying its value to x.

We contend that the enveloped representation of functions as described in this section is useful for understanding lexical binding in programming languages, and the accompanying issues, e.g. the "funarg" problem [Moses 70].

2.7.3. Productions can be eliminated.

We now wish to show how the enveloping device can be used to eliminate the need for productions. Although productions are a useful representation for gaining intuitive understanding, they are awkward for representing the idea of imported values, since all such values would presumably have to come from a single context. In our "block-structured" implementation of FGL [Keller, et al. 80] we have found it convenient to abandon the implementation which corresponds most closely to productions, in favor of one which treats all programmer-defined functions uniformly, whether or not they are returned as values.

Consider a graph grammar production of the form shown in Figure 2-19. We can view the consequent of the above production as an abbreviation for the

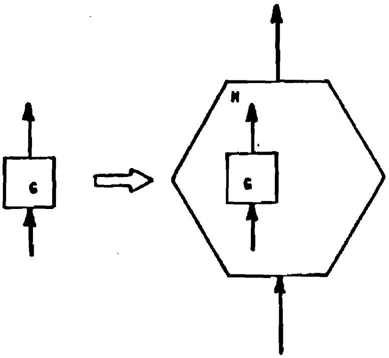


Figure 2-19: Typical recursive production

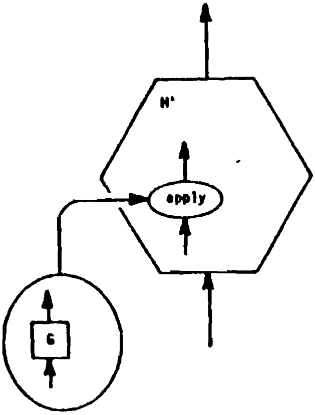


Figure 2-20: Equivalent of the consequent of the production of Figure 2-19 using apply

subgraph shown in Figure 2-20, where H' is like H, except that G has been replaced with the apply as shown.

In other words, H' is a "functional" which takes an argument which can be supplied as G, whereas H has G built in. Thus, we could write

$$H = H'(G).$$

This equation will find additional uses later.

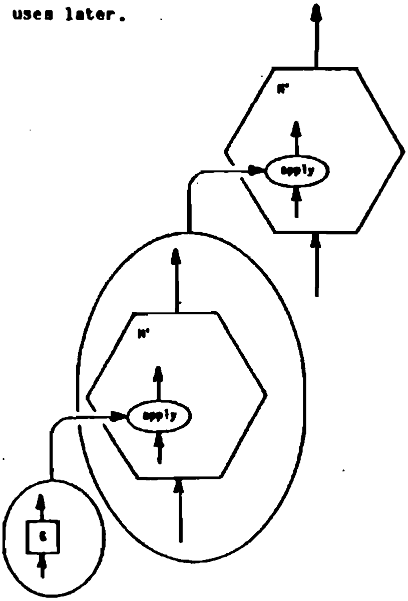


Figure 2-21: A second graph equivalent to the G of Figure 2-19

Since the graph of Figure 2-20 is equivalent to the function G, we may substitute the entire graph for G, as shown in Figure 2-21. By folding the graph in this figure, effectively using the equation

$$G = H'(G),$$

we get an equivalent but more compact version, as shown in Figure 2-22, as well as a further simplified version in Figure 2-23. The latter can always be used in place of G itself.

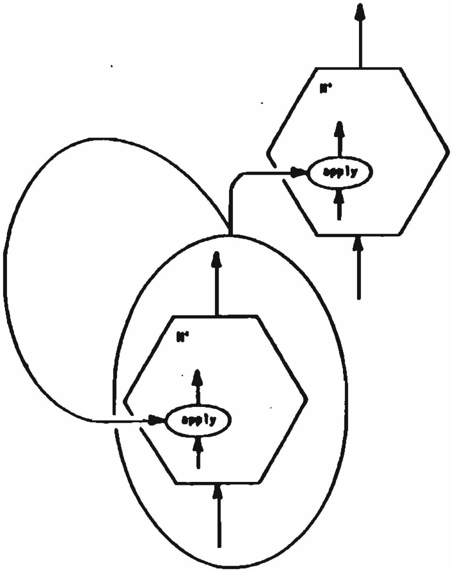


Figure 2-22: Folded version of G

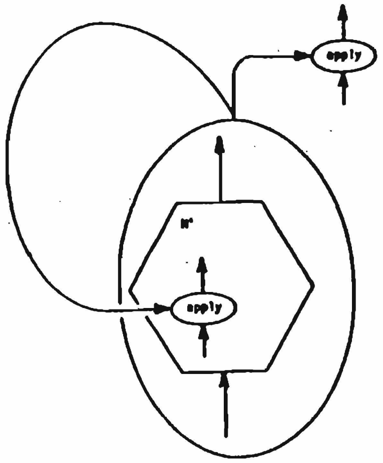


Figure 2-23: Simplified folded version of G

3. Theoretical Basis

3.1. Function Graphs and Equations

We now deal with the problem of determining the long-range behavior of a general network. We have not yet provided any reason to believe that this behavior is unique in any sense, particularly in the context of asynchronous concurrent computation. Sufficient conditions for this uniqueness will be provided in the course of the presentation, which is at the general function graph level.

As mentioned earlier, a graph consists of a set of node functions which act on arc data structures. It has already been seen how such a graph could be characterized by a system of equations involving the arcs as variables. We now wish to represent all node functions collectively as one function F acting on a tuple of data structures. Correspondingly, the system of equations will be reduced to a single equation.

3.1.1. One function and one equation suffice.

We call F the system function of the graph. It intuitively gives that segment of the overall behavior corresponding to one step of all node functions acting in concert without feedback. The components of the tuple on which F acts correspond to the "internal" and "output" arcs of the graph, with the "input arcs" of the graph as "parameters" of F. By input arc, we mean one which is not directed out of any node in the graph, and by output arc, we mean one which is not directed into any node. An internal arc is one which is neither an input nor an output arc.

3.1.2. Example:

Consider the graph of Figure 3-1. Here the output arc y is already identified with an internal arc z_1. x_1 and x_2 are the two input arcs. We express the system function F in terms of f, g, and h by

$$F(z_1, z_2, z_3) = (f(x_1, z_2), g(x_2, z_3), h(z_1, z_2))$$

As mentioned, F implicitly depends on the inputs x_1 and x_2. The solution of

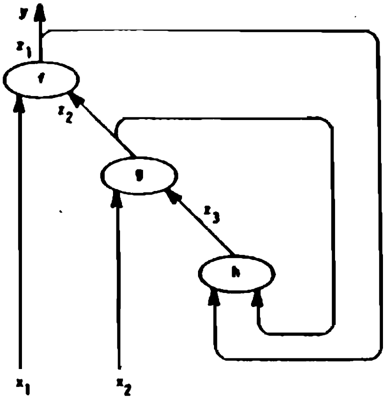


Figure 3-1: Function graph example

3.2. Data Types

We now present conditions under which the system function determines most of the relevant aspects of the graph's long-term behavior. We first require a means of characterizing how data structures are built up over an interval of observation. Formally, this amounts to requiring that our data structures be members of the "domain" of a "data type". (Although the phrase "data type" may appear contradictory to the popular term "abstract data type", we shall use it to have a meaning in the sense of [Scott 70], which is still the prevalent use of the term in the area of semantics of computation.) Although the generality in this section may appear to be overkill, it has genuine value in understanding the scope of the theory behind concurrent execution of function graphs and what can be proved with them.

Definition A data type consists of:

(i) a set D, called the domain of the data type,

(ii) an information ordering < on D,

(iii) an *undefined element* ? in D, and

(iv) a *limit operation* lim.

More specifically, the information ordering is a *partial order* on the domain D, i.e. a binary relation which has the following properties:

(i) anti-symmetric: For all x, y in D,

$$x < y \text{ implies not } y < x$$

(ii) transitive: For all x, y, z in D,

$$(x < y \text{ and } y < z) \text{ implies } x < z$$

When we say that ? is an undefined element, we mean that it is the unique element such that

For all x in D-{?},

$$? < x$$

3.2.1. Data types characterize information content.

The information ordering provides a way of *comparing the information* in two data structures. Thus, if x and y are two possible structures, x < y means that y contains more information than x. ? is sometimes called *bottom*. It represents a structure about which there is *no* information.

For notational convenience, we extend our notation for the ordering < to $\leq$, in the sense that

$$x \leq y \text{ means } x < y \text{ or } x = y$$

Conversely, given such an ordering $\leq$, we can recover < by defining x < y to mean $x \leq y$ and $x \neq y$.

Finally, we define the notion of a *limit operation*. If C is a subset of D and d an element of D, we write

$$C \leq d$$

if for all x in C, $x \leq d$. In this case, we say that d is an *upper bound* on C.

If d is such that $C \leq d$ and for every d'

$$C \leq d' \text{ implies } d \leq d'$$

then we say that d is the least upper bound of C, or limit of C. That is, d is an upper bound on C which is $\leq$ every upper bound of C.

When such a limit exists, we shall denote it as a function of C by

$$\lim C$$

In a data type, we require that lim C exist whenever C is a "chain", which means

For all x, y in C,

$$x \leq y \text{ or } y \leq x$$

In other words, a chain is a set wherein any two distinct members can be ordered with respect to the amount information in them. The limit of the chain corresponds to the information contained in all of the members of the chain, and no more.

3.2.2. Limits epitomize successive approximations.

It makes sense to require that the data structures appearing on arcs be members of the domain of data types associated with those arcs. The information ordering determines which data values can appear consecutively on an arc; i.e. we require $x < y$ whenever y appears after x. In a sense, this says that x is an approximation to y.

Furthermore, we can identify the ultimate structure appearing on an arc as the limit of the set of successive approximations appearing there. This provides a convenient way of characterizing behavior even in the case where such behavior is non-terminating.

3.2.3. Example of a data type:

The handshaking example in Section 2.4.2 deals with data types having domains of sets of strings over some alphabet, including infinite strings. We cell these strings one-level streams, to contrast with a more comprehensive type of stream to be discussed subsequently. The undefined element in the domain corresponds to the null string. The information ordering coincides with the prefix ordering. The limit of a chain of strings is just the shortest string having all strings in the chain as prefixes. For example,

$$\text{lim } \{?, b, bb, bbb, \ldots\} = bbb\ldots$$

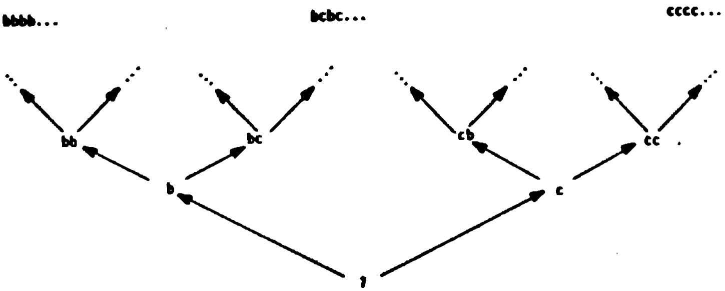


Figure 3-2: Ordering diagram for a one-level stream data type

The information ordering in a data type can be depicted by an ordering diagram which shows how typical members relate to one another. In such a diagram, if there is an arrow from x to y, then $x < y$ in the information ordering. Transitive arrows are not shown explicitly. In other words, $x < y$ also if there is a sequence of arrows directed from x to y. The ordering for one-level streams over the alphabet $\{b, c\}$ is shown in Figure 3-2.

3.2.4. Examples of data types:

1. Let S be any set. Then $P(S)$, the set of all subsets of S, is the domain of a data type having least element $\emptyset$ (the empty set), information ordering $\subseteq$ (set inclusion), and limit operation $\cup$ (union). Shown in Figure 3-3 is the ordering diagram for $P(S)$ where S is the set of all natural numbers, $\{0, 1, 2, 3, \ldots\}$.

2. Let S be any set. Then $B(S)$, the set of all bags, i.e. "sets with possibly repeated elements", of members of S, with inclusion and union as in (1), forms a data type. Shown in Figure 3-4 is the ordering diagram for $B(S)$ where S is the set of two atoms $\{a, b\}$.

3. Let S be any set. Let ? be an element not in S. Let the domain of the data type be $S \cup \{?\}$ with ordering $<$ defined by

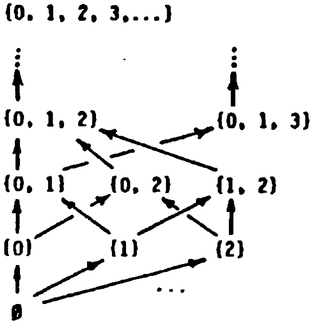


Figure 3-3: P(S) ordering

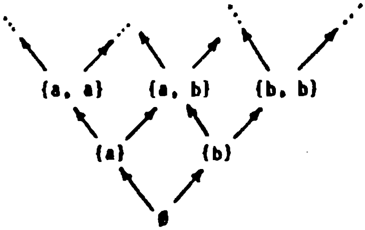


Figure 3-4: B(S) ordering

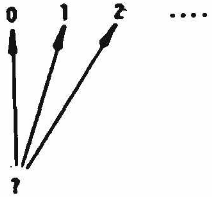


Figure 3-5: Flat ordering of the natural numbers

$$x < y \text{ iff } (x = ? \text{ and } y \neq ?)$$

This is called the *flat data type* over S. Notice that each chain in such a data type has at most two members and the limit is just the greater of the two. Shown in Figure 3-5 is the flat data type on the natural numbers.

4.

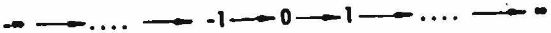


Figure 3-6: numeric ordering

Let Z be the set of all integers. Then $Z \cup \{\infty, -\infty\}$ is the domain of a data type with the information ordering of numeric inequality ($\leq$) and maximum as the limit operation. This ordering is demonstrated in Figure 3-6.

5. Let S be a set, called the set of *atoms*. We define a data type whose domain is the set of *binary trees* over S. Begin by defining the *finite* binary trees:

   (i) The null tree, ?, is a finite binary tree.

   (ii) Any member of S is a finite binary tree.

   (iii) If t_1 and t_2 are finite binary trees, then so is the tree (t_1, t_2) having t_1 as its left subtree and t_2 as its right subtree.

The ordering $\leq$ on finite binary trees is defined by:

(i) $? < t$, for each $t \neq ?$

(ii) $(t_1, t_2) \leq (t_3, t_4)$ iff $t_1 \leq t_3$ and $t_2 \leq t_4$.

We then define the <u>infinite binary trees</u> to be limits of infinite chains of finite binary trees. Thus a binary tree is either a finite binary tree or an infinite binary tree.

For example, the rules above tell us that

$$? < (?, ?) < ((?, ?), ?)$$

Extending this construction, we have

$$t_0 < t_1 < t_2 < \ldots$$

where $t_0 = ?$ and for each i, $t_{i+1} = ((?, t_i), ?)$.

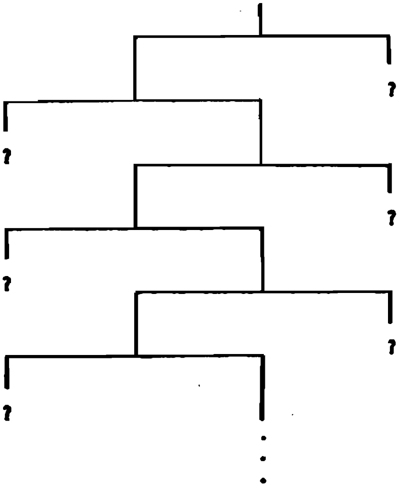


Figure 3-7: Limit of the tree sequence $t_0 < t_1 < t_2 < \ldots$

The limit of this infinite sequence is the infinite tree depicted in Figure 3-7.

We shall observe an important application of the binary tree data type in a forthcoming section. It can be noted at this point that the binary tree data type can also be viewed as a <u>multi-level</u> <u>stream</u> data type over a set of atoms, wherein we define

(i) The null stream ? is a stream.

(ii) Each atom is a stream.

(iii) Any finite or infinite sequence of streams is a stream.

The corresponding ordering is

(i) $? < x$ for all $x \neq ?$

(ii) $x \leq y$ iff x is not longer than y and each component of x is $\leq$ the corresponding component of y.

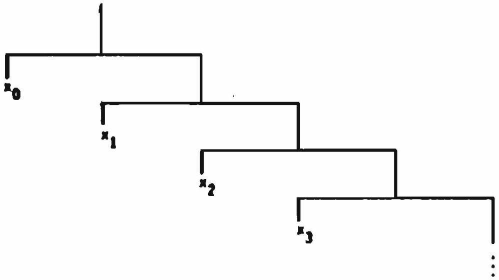


Figure 3-8: The tree equivalent to the stream $x_0, x_1, x_2, \ldots$

The connection with binary trees is that the stream

$$x_0, x_1, x_2, \ldots$$

is equivalent to the tree shown in Figure 3-8. This connection is used in languages such as Lisp, which sometimes use a special atom 'nil' as the leaf of a tree to indicate the end of a finite stream which cannot be further extended. It is important not to confuse 'nil' with the null stream ?.

3.2.5. Data types combine to get new date types.

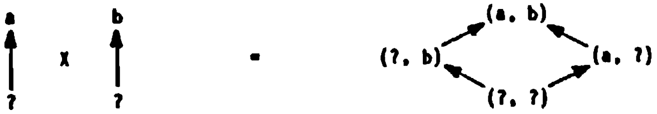


Figure 3-9: Product data types

It is important to notice that if we have a collection of data types, D_i, $?_i$, $<_i$, lim_i, then we may form their <u>product</u> <u>data</u> <u>type</u>

1. $D = X D_i$,
2. $? = (?_1, ?_2, ...)$,
3. $<$ is defined by

   $(d_1, d_2, ...) \le (d_1', d_2', ...)$

   iff

   for each i, $d_i \le_i d_i'$

and extending the limit operation so that

$$lim[(d_1, d_2, ...), (d_1', d_2', ...), (d_1'', d_2'', ...), ...]$$

$$= (lim_1[d_1, d_1', d_1'', ...], lim_2[d_2, d_2', d_2'', ...], ...)$$

We illustrate product data types in Figure 3-9.

To summarize our interest in the notion of data types, we require that the data structures representing the history of an arc in a graph be members of a data type. The information ordering of a data type constrains the transitions between histories of any arc. That is, a data structure x can be later followed by a structure y only if $x \le y$. The <u>limit</u> requirement of a data type provides for the existence of a unique (possibly infinite) "ultimate" structure on any arc of a function graph.

3.3. Behavioral Descriptions

Returning to the handshaking example of Figure 2-8, we further clarify the discussion by pointing out that there are two viewpoints for the behavioral description given. First, we reduce the discussion to one equation. We had

$$\begin{aligned} x &= \text{left}(y) \\ &= b\ \text{invert}(y) \\ &= b\ \text{invert}(\text{right}(x)) \\ &= b\ \text{invert}(\text{invert}(x)) \\ &= b\ x, \text{ since } \text{invert}(\text{invert}(x)) = x. \end{aligned}$$

There are two essential ways of viewing an equation such as

$$x = bx$$

The first is the view that the behavior at any step is given by following the behavior so far by b and continuing. This is suggested by the repeated substitution for x, viz.

$$\begin{aligned} x &= bx \\ &= bbx \\ &= bbbx \\ &= . \\ &\ . \\ &\ . \end{aligned}$$

The second is the view that the ultimate behavior x is obtained by successive approximations, starting with the undefined behavior, as in

$$\begin{aligned} x &= ? \\ x &= b? \\ x &= bb? \\ &. \\ &. \\ &. \end{aligned}$$

To more accurately describe the method of successive approximations for determining system behavior, we represent the node functions of the graph by the system function F on the product of the data types at each internal arc. Recall that the input arc data values are implicit parameters of F.

Assume for now that the arcs of a given graph are initialized so that the input arcs contain the ultimate values to be placed on those arcs by the environment and the internal arcs are initialized to contain the 'undefined' structure, ?. It turns out that no generality is lost in these assumptions.

3.3.1. Restrictions are necessary for successive approximations to work.

In order to insure the efficacy of the successive approximation approach, we shall place some requirements on F. The first requirement is that of monotonicity. To say that F is monotone means

For all d, d',
    if d $\leq$ d' then F(d) $\leq$ F(d').

In particular, from the "seed" relationship

$$? \leq F(?)$$

we may apply monotonicity repeatedly to get

$$F(?) \leq F(F(?))$$
$$F(F(?)) \leq F(F(F(?)))$$
$$\vdots$$

In short, we have a chain

$$\{?,\ F(?),\ F(F(?)),\ \ldots\}$$

By our assumptions about data types, this chain has a limit, which we henceforth denote by

$$F^{\infty}(?)$$

Notice that the chain above corresponds to the "simulation" of only one of what might be many possibly computations. No assumptions have been stated about relative computation times of the node functions, but this one simulation assumes that they complete each step synchronously.

3.3.2. Monotonicity insures speed-independence in an asynchronous environment.

Fortunately, the monotonicity property insures that $F^{\infty}(?)$ is always the result, <u>independent</u> of the manner of simulation. It is only required that each node eventually realizes the value specified by its function applied to its ultimate input data structure. In this case, each step of an arbitrarily-timed computation will eventually be subsumed in the limit value. In other words, the result of the computation is <u>determinate</u> or <u>speed-independent</u>. (This is also related to the "Church-Rosser" property, cf. [Rosen 73].)

An additional restriction must be imposed to insure that $F^{\infty}(?)$ "makes sense" as an ultimate behavior. The following section elaborates on this point.

3.4. Continuity

Although $F^{\infty}(?)$ is interpretable as the unique behavior of the function graph, it does not necessarily follow from the properties described so far that $F^{\infty}(?)$ is a fixed point, i.e. it <u>satisfies the system equation</u>,

$$F(F^{\infty}(?)) = F^{\infty}(?)$$

The inequality

$$F^{\infty}(?) \leq F(F^{\infty}(?))$$

follows from monotonicity, but the converse inequality

$$F(F^{\infty}(?)) \leq F(?)$$

does not.

In other words, there is no guarantee that $F^{\infty}(?)$ is "stable", in the sense that it indeed represents the ultimate value which the function F is "trying to produce". One can easily construct examples consistent with all properties introduced so far which show that the above fixed point property does not hold. For instance, let

$$h(x) = \begin{cases} \text{cons}(a,\ x) & \text{if } x \text{ is a finite stream} \\ \text{cons}(b,\ x) & \text{if } x \text{ is an infinite stream} \end{cases}$$

Then h is monotone, since if x is a prefix of y then cons(a, x) is a prefix of cons(a, y), and similarly with a replaced by b. However, h does not satisfy the system equation, since

$$h^{\infty}(?) = \mathrm{cons}(a,\ \mathrm{cons}(a,\ \mathrm{cons}(a,\ \ldots)))$$

but

$$h(h^{\infty}(?)) = \mathrm{cons}(b,\ \mathrm{cons}(a,\ \mathrm{cons}(a,\ \ldots)))$$

Although there is no great mathematical harm in not having the above system equation hold, without it we would have that the anomaly that our system function F could be applied to the limit of the chain (i.e. the ultimate behavior) to get _new information_ not present in the chain itself, which seems counter-intuitive to physical reality. A sufficient condition on F which results in the stability of $F^{\infty}(?)$ is that of _continuity_.

The function F:D --> D is called _continuous_ provided that it is monotone and for any chain $C \subseteq D$,

$$F(\lim C) = \lim \{ F(d) \mid d \text{ in } C \}$$

(Notice that monotonicity insures that the set on the right is a chain, so that it makes sense to consider its limit.) By identifying

$$\{?,\ F(?),\ F(F(?)),\ \ldots\}$$

with C and noting that

$$\lim \{?,\ F(?),\ F(F(?)),\ \ldots\} = \lim \{F(?),\ F(F(?)),\ \ldots\},$$

we get $F^{\infty}(?)$ as a fixed point.

For example, it is easy to see that the functions left and right from the handshaking example are both continuous, so that the derived limit is the least fixed point.

3.4.1. Example:

Consider the binary tree data type introduced earlier. It is easy to see that the functions head, tail, and cons, defined as follows, are all continuous:

```
cons(x,y) = (x, y)

head((x,y)) = x
head(?) = ?
head(a) = error, if a is an atom

tail((x,y)) = y
tail(?) = ?
tail(a) = error, if a is an atom
```

Here error is a special value which is distinguished from all other values and indicates that a "non-sensical" application of a function has been attempted. Notice that error is quite distinct from ?, the latter being the mathematical value indicating the result of a divergent or incomplete computation. Notice that under the stream interpretation of trees, head corresponds to the *first* member of the stream, while tail corresponds to the *rest* of the stream after deleting the first member.

3.4.2. Determinacy Theorem

We now encapsulate the essence of the above discussion in a theorem.

> *Determinacy Theorem* If G is any function graph composed of nodes which represent continuous functions on their connecting arc data types, then G determines a unique function from the data types of its input arcs to those of its output arcs. Moreover, if each node function ultimately realizes its output value on its ultimate input values, then G also will realize its output value.

The subtlety of this theorem is that the input to a given node may well by a "moving target", i.e. its input may be changing, since that input value might be in the process of being produced by some other node, whose input may be changing, etc. Continuity insures that despite such mobility of values, a least (with respect to the information ordering) tuple of arc values consistent with the specified functions exists. This tuple is the *least fixed point* of the system of equations. It corresponds to the solution of the system which requires introduction of no additional information except that exhibited in the functions and equations themselves (e.g. no information concerning the method of evaluation). Successive approximations give us one way of ascertaining that tuple.

3.4.3. Example:

Using the binary tree data type, consider the equation

$$z = cons(x, z)$$

We have already mentioned that cons is continuous on this data type. For successive approximations to z we get

$$cons(x, ?)$$

$$cons(x, cons(x, ?))$$

$$cons(x, cons(x, cons(x, ?)))$$

•

•

The least fixed point is apparently the infinite structure

$$z = cons(x, cons(x, cons(x,...)))$$

Clearly, the equation is satisfied when this structure is substituted for z in z = cons(x, z).

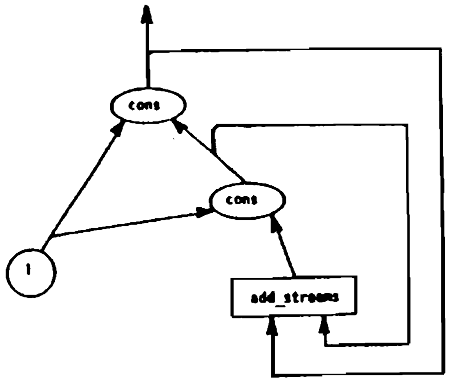


Figure 3-10: Graph resulting in the Fibonacci stream

3.4.4. Example:

The graph in Figure 3-10 produces the stream of Fibonacci numbers. To see this, we may use the successive approximation technique.

The system function is given by

F(x, y, z) = (add_streams(y, z), cons(1, x), cons(1, y))

so that we have the following successive approximations to (x, y, z):

(?, 1 ?, 1 ?)

(2 ?, 1 ?, 1 1 ?)

(2 ?, 1 2 ?, 1 1 ?)

(2 3 ?, 1 2 ?, 1 1 2 ?)

(2 3 ?, 1 2 3 ?, 1 1 2 ?)

(2 3 5 ?, 1 2 3 ?, 1 1 2 3 ?)

(2 3 5 ?, 1 2 3 5 ?, 1 1 2 3 ?)

(2 3 5 8 ?, 1 2 3 5 ?, 1 1 2 3 ?)

.
.
.

The limit is $F^{\omega}(?)$ =

(2 3 5 8 13 21 34...., 1 2 3 5 8 13 21...., 1 1 2 3 5 8 13...)

Another way to motivate the choice of $F^{\omega}(?)$ as the behavior is to use the aforementioned notion of *repeated substitution* in the equation x = F(x). That is, by repeatedly substituting the right-hand side for the left, we get

x = F(F(F(...)))

This solution agrees with the successive approximation solution.

3.4.5. Continuity insures composability.

An additional advantage which accrues from assuming that the node functions of a network are continuous is that a closure property is easy to demonstrate. A useful technique in system structuring is to treat a system as if it were composed of sub-systems, rather than of atomic node functions. It would then be useful to know that such sub-systems behaved essentially as if they were atomic nodes. We can show that continuous functions are closed under functional composition, so that continuity of individual node functions insures continuity of the system function. Such a property is important in hierarchical and modular development of software and hardware systems.

Note that arbitrarily many identity functions may be inserted on any arc of a function graph composed of continuous functions, without affecting the ultimate function computed. Therefore, these graphs exhibit what is called delay-insensitivity [Keller 74], in that the identity functions act as arbitrary inserted delays. When delay-insensitivity holds for a distributed system, it tends to be much easier to analyze than in the more general case.

A further ramification of continuity is discussed in Section 3.6.

3.5. Determinacy of Systems involving Productions

We now wish to extend the closure property discussed above to allow auxiliary nodes as well. That is, given a graph grammar, if each terminal node represents a continuous functien, then so does an arbitrary graph.

3.5.1. A set of functions may be a data type.

Some preliminary observations will aid us. First, let D_1 —> D_2 denote the set of continuous functions from D_1 into D_2, where D_1 and D_2 are the domains of two data types. Then D_1 —> D_2 itself is the domain of a data type, the ordering of which is defined by

$F \leq G$ if, and only if,

for each x in D_1, $F(x) \leq G(x)$.

The least element ? of this type is the function whose value is always the least element of D_2. The limit operation is defined so that for any chain F_1, F_2, F_3, ... in $D_1 \longrightarrow D_2$,

for each x in D_1,

$$(\lim \{F_1, F_2, F_3, \ldots\})(x) = \lim \{F_1(x), F_2(x), F_3(x), \ldots\}$$

Referring to the graph of Figure 2-23, which represents the definition of a function G according to $G(x) = H'(G)(x)$, as discussed in Section 2.7.3, we may use splitting to unfold the graph into numerous infinite forms, three of which are shown in Figure 3-11. The point of these foldings and unfoldings, besides being an exercise in graph manipulations, is that the infinite form Figure 3-11b shows that the recursively-defined function G represents the function

$$H'^{\infty}(?) = \lim \{?, H'(?), H'(H'(?)), H'(H'(H'(?))), \ldots\}$$

where H' is the function represented by the consequent in Figure 2-20 and ? represents the function whose value is totally undefined. The infinite form of Figure 3-11c is the equivalent of repeated substitution and gives another representation, namely

$$H'^{\infty}(?) = H'(H'(H'(\ldots)))$$

It is not difficult to show that the limit function above is continuous, thereby allowing us to conclude the following extension of the determinacy theorem:

> Recursion Theorem Any function graph with continuous atomic functions, including one with auxiliary nodes defined by productions, itself represents a continuous function. This function is determined by the graph formed by repeated substitutions of antecedent nodes by their corresponding consequents.

It is noted that the limit concept in our notion of data type is essential in making the above statement meaningful, since this concept gives meaning to the function represented by an infinite graph.

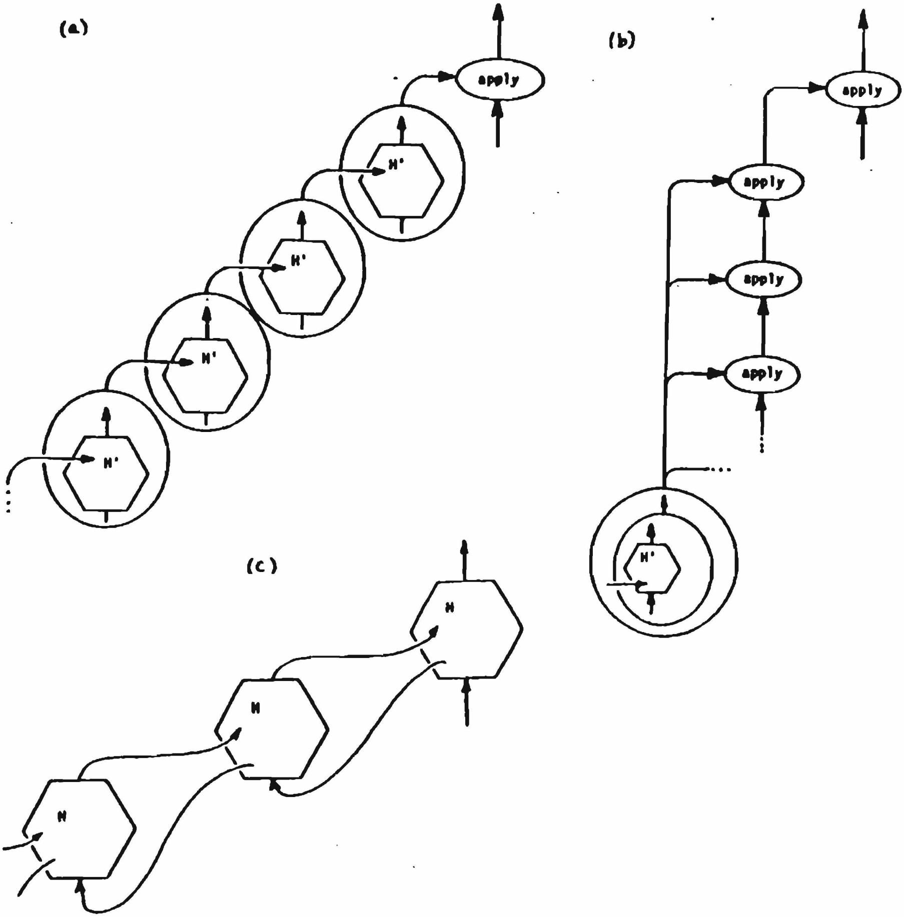


Figure 3-11: Infinite unfolded versions of G

3.6. Finite Support

Continuity has another interesting implication. Consider the output data structure generated by a node function. This structure may, in the limit, be infinite. However, we expect that it will always be generated incrementally, by a succession of finite approximations. Correspondingly, we would expect that each finite approximation be the result of the node function's action on a finite approximation to its input, rather than an infinite amount of input. This suggests that the set of data structures D which comprise a data type be dichotomized into the set of "finite" structures D_{fin} and the "infinite" structures D_{inf}, and that we have the following finite support condition:

For each d in D, if F(d) is in D_{fin},
    then for some d' in D_{fin},

$$d' \leq d \text{ and } F(d') = F(d).$$

The distinction of D_{fin} vs. D_{inf} depends on the data type under consideration. It is clear for strings and trees, but perhaps not so clear in general. One proposed definition for general data types (which places an additional constraint on the ordering $\leq$) is suggested in [Stoy 77], pages 106-111, but to explore this suggestion would exceed the scope of this paper.

Here we shall be content with an example, showing that the finite-support property holds for continuous functions on strings. Suppose that d is a (possibly-infinite) string such that F(d) is finite. Let us write d as the limit of the finite strings

$$d_1, d_2, \ldots$$

By continuity, F(d) is the limit of

$$F(d_1), F(d_2), \ldots$$

But since F(d) is a finite string, there must be an i such that

$$F(d) = F(d_i)$$

so choose $d' = d_i$. The finite support property therefore holds.

4. Machine evaluation of Computations Represented by Graphs

Having presented examples of the use of graphs for specification, it is now time to discuss the evaluation of functions specified by them. That is, given a function graph with data objects specified on each of its input arcs, by evaluation we mean the procedure to be used to cause ultimate production of the data objects on the output arcs of the graph.

4.1. A Rudimentary form of FGL

As we have presented a rather abstract formulation of the semantics of function graphs, the notion of evaluation will clearly be dependent on the choice of underlying data types and atomic operators. Hence, we can at best hope to present an evaluation method for an exemplary choice of the latter. This choice will be a simple language which we call FGL.

4.1.1. The data types of FGL

In this presentation, the set of data objects of FGL will be

Objects = Atoms U Tuples U Graphs U {?}

where

1. Atoms = Integers U Strings U {error}, where Integers is the set of integers and Strings is the set of character strings over some alphabet. We assume that Strings includes the string 'nil' which will play the role of the Boolean value false. Any atom other than 'nil' and {error} may play the role of the Boolean value true.

2. Tuples: A tuple is a sequence of N Objects. (In Lisp, N = 2 is typically required.)

3. Graphs: We allow the enveloping of a graph, as described in Section 2.5, and its use as a function closure data object.

As we wish these objects to comprise the domain of a data type, we must supply ordering and limit information accordingly. First, there is a least element ?, representing a value which has not yet been determined. For each data object x ≠ ?, we have ? < x.

Second, each atom is unordered with respect to every other. Third, the ordering between two tuples is given by

$$(x_1, \ldots\ldots, x_n) \leq (y_1, \ldots\ldots, y_n)$$

iff for each i, $x_i \leq y_i$.

Finally, if G and H are graphs, then G < H iff H is like G, except that some substitutions have been made for auxiliaries in G to obtain H.

4.1.2. Basic Operators of FGL

We now describe a basic set of FGL operators. The expected types of arguments are indicated by the following names.

obj	an object of any type
tup	a tuple
int	an integer
bool	a string which is either 't' or 'nil'
fun	an enveloped graph, representing a function closure

The logical functions and and or consider 'nil' to be false and everything else to be true. The following descriptions use the words true and false in place of the strings 't' and 'nil'.

The forms in the following descriptions indicate the expected argument types, followed by a colon, followed by the result type. For single argument functions, parentheses may be omitted. Violations of the expected type of an argument will result in the special value error. It is assumed that if a function has error as the value of one of its required arguments, then the result of that function will be error.

name	form and meaning
add	int$_1$ + int$_2$: int Adds two numeric arguments.
and	obj$_1$ and obj$_2$: bool The logical conjunction of its arguments. and is sequential, evaluating the second argument only if the first is false.
apply	fun$_1$(obj$_2$, obj$_3$, ..., obj$_n$): obj, if fun$_1$ is a variable (not a general expression), or apply(fun$_1$, obj$_2$, obj$_3$, ..., obj$_n$): obj generally. Applies first argument to remaining arguments.
atom	atom (obj$_1$): obj true unless argument is a tuple.

head

head(tup$_1$): obj
First component of a tuple argument.

tail

tail(tup$_1$): obj
Last component of a tuple argument.

cond

if obj$_1$ then obj$_2$ else obj$_3$: obj
Evaluates obj$_1$. If the result is not false, then the second argument is returned, otherwise the third argument is returned.

cons

cons(obj$_1$, obj$_2$, ..., obj$_n$): tup
Forms a tuple of its arguments, of which there may be any number.

eq

obj$_1$ = obj$_2$: bool
true if arguments are atoms and have the same value.

lessp

int$_1$ < int$_2$:bool
Returns true if the numeric first argument is less than second, and false otherwise.

mod

int$_1$ mod int$_2$: int
First integer argument modulo second.

mult

int$_1$ * int$_2$: int
Product of two numeric arguments.

null

null(obj$_1$): bool
Returns true if argument is false, returns false otherwise. (Use this for logical negation.)

numberp

numberp(obj$_1$): bool
Returns true if argument is numeric, false otherwise.

or

obj$_1$ or obj$_2$: bool
Logical disjunction of its arguments, evaluating second argument only if first is false.

select

select(int$_1$, tup$_2$): obj
Gives the component indexed left to right by int$_1$ of the tuple object tup$_2$. The components are indexed 1, 2, ..., n. If int$_1$ is negative, then indexing is right to left by -1, -2, ..., -(n-1), -n. An error results if int$_1$ is 0 or out of bounds. (head and tail correspond to select(1, ...) and select(-1, ...), respectively.)

4.1.3. Internal representation of FGL

We present a form of acceptable for storage in computer memory. For sake of concreteness, assume a conventional linearly-addressable memory. We concentrate on the representation of a single graph within this memory.

Assume for simplicity that the memory has a word-size large enough to store all the information required about a single FGL node. If this is not the case, multiple word encodings may be used. The information stored includes an encoding of the name of the function, and the references to arguments to the function. Since nodes are stored one per word, we identify the address of the word containing the node with the arc leaving that node. Therefore the references to the arguments of a node are just the addresses of the nodes which produce those arguments as their result.

All of the address information described above can be made relative to a block of words which contains the encoding of all nodes for a single graph, normally the consequent of a production or the contents of an envelope. The base address of this block can then be identified with this graph, and used as the argument to an apply. To be more precise, a closure must be accompanied by a tuple of import values, as well as the base address of the block, for those imports may be different for each instantiation of the enveloped graph.

```
DEF SUMSTREAM
RESULT ARC_4
ARGUMENTS X
IMPORTS ADD_STREAMS
ARC_1 CAR X
ARC_2 CDR X
ARC_3 APPLY ADD_STREAMS ARC_2 ARC_4
ARC_4 CONS ARC_1 ARC_3
ENDDEF
```

Figure 4-1: Assembly language encoding of an FGL production, that of Figure 2-1.

We could then proceed to give an "assembly language" version of FGL. A block of code is represented by a sequence of "lines", where each line encodes one node of the block. A line contains a symbolic label for the corresponding node, followed by the name of the function, and the labels of the arguments to the the function. Shown in Figure 4-1 is the assembly code for a simple FGL graph.

4.2. Evaluation

We shall describe a destructive form of evaluation, in which the nodes of the graph are replaced with their values. This means that for each use of an apply, the block which encodes the closure will have to be copied afresh. This copying supplants the usual initialization which must accompany procedure entry, etc.

We have already explained how each graph with objects specified on its input arcs determines a unique tuple of objects on its output arcs. We must now describe an evaluation mechanism which insures that the mathematically-determined values do get produced.

Since objects are abstract, i.e. we have not really defined what it means to produce an object, we can content ourselves with a primitive which produces a single atom, say by printing it on a line printer. We can then use this primitive to display general objects in whatever image of their abstract form we feel appropriate, by displaying the atoms which comprise these objects. For example, if we want to print a tuple in the form with parentheses and commas, then we could do so by applying our print primitive to strings consisting of parentheses, commas, and the atoms which comprise the tuple. Since the result objects might well be infinite, it seems prudent that we produce parts of objects by demand.

To continue with our discussions of demand production of objects, assume that there is an abstract entity known as "demand" which can be present on any arc of a function graph. This entity remains on the arc until it is satisfied by the presence of a predetermined portion of the object. For the current language, we convene that this portion must be either an atom, a graph, or a skeletal tuple, i.e. a tuple having the number of its components, but not necessarily the components themselves, specified.

Prior to the demand being satisfied, the arc value is ?, at which it may remain forever if the demand is never satisfied. We intend for the latter to happen only if the ultimate functionally-determined value is ?.

4.2.1. Description of demand propagation

To complete the specification of the evaluation process, we must specify how the demand is propagated through each of the atomic operators.

cons When the result of a cons is demanded, the demand is immediately satisfied by making the result a skeletal tuple, the length of which is the number of input arcs to the cons. It suffices to have the cons node itself play the role of the skeletal tuple, so no actual replacement is necessary. Demand does not propagate to the components themselves until as specified in select below.

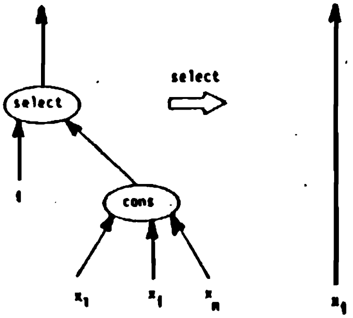


Figure 4-2: Demand evaluation of select

select When the result of select(i, x) is demanded, demand propagates to both arguments. When both of the latter demands are satisfied, if n is the number of components of x and $1 \leq i \leq n$, then the select is deleted, its output arc being connected directly the i^{th} skeletal argument. Demand remains, and is propagated to that argument. The diagram of Figure 4-2 is meant to be suggestive.

atom The demand propagates to the argument. When the argument demand is satisfied, the output arc gets the appropriate logical value.

cond The demand propagates to the first argument. When that demand is satisfied, the output arc is connected to the second or third input arc, depending on whether or not the value of the first argument is 'nil', then demand propagates to the chosen argument.

eq Demand propagates to both arguments. When both are satisfied, the function is evaluated and the result appears on the output arc of the node.

apply(G, x) Demand propagates to argument G. When that demand is satisfied, a copy of graph G is made, with the free input arc connected to the argument x. The output arc of the copy replaces the apply node, and demand propagates to the output arc of the copy. Thus the apply rule of Figure 2-16 is mimicked.

Binary arithmetic operators propagate demand like eq. The propagation of demand in other operators not listed above may be inferred from the propagation for those listed. Figure 4-3 illustrates the propagation of demand through a complete, but very simple, example.

4.2.2. Correctness of an FGL evaluator

The correctness of an evaluator can be stated informally as follows:

> For any arc at which a demand presents itself, if the value determined by the function is an atom, then the value ultimately appears on that arc.

Now consider any evaluator having the property that for any arc on which a demand eventually appears, the evaluator eventually treats the arc according to the specifications for demand/value propagation. This property may be insured, for example, by a combination of task-list and notifier structures [Keller and Lindstrom 80].

We claim that such an evaluator is correct as stated above. We do not go into further formalization or proof of this claim here, except to say that it is naturally conducted by induction, based on the depth of a demanded sub-object within the overall result object. A more complete proof appears in [Keller and Lindstrom 80]. Proof sketches for related models may be found in *[Friedman and Wise 76]* and *[Henderson and Morris 76]*.

4.2.3. Parsimonious evaluation

Another property attributable to the mode of computation described here is parsimonious evaluation, i.e. that a value appearing on an arc which fans out only need be computed once. This is accomplished by simply keeping track of whether an arc's value has been demanded and not propagating any but the first demand. When and if a value finally arrives at that arc, it is available to

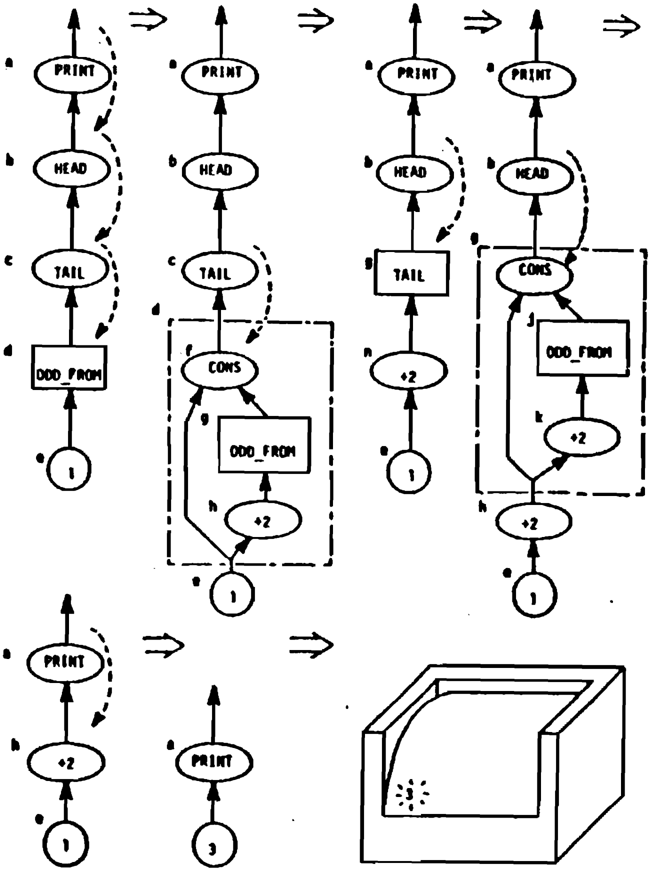


Figure 4-3: Illustration of demand propagation (shown by dashed lines). odd_from is defined in Section 2.1.2.

all operators which had demanded it, as well as those which will demand it in the future. This implementation technique has found use in linking mechanism in operating systems, e.g. Multics [Organick 72]. It has been called by the term "suicidal suspension" in [Friedman and Wise 76], because the "suspensions" (i.e. the encodings of node functions) kill themselves by replacing there code with the value of the function.

4.3. A Higher Level FGL

As described earlier, an FGL graph may be encoded in a form of "assembly language". However it would be quite tedious to program and read extensive examples in such a language. For this reason, it is worth pursuing higher level textual representations of FGL. One candidate representation, called Textual FGL, which has been implemented by the author and colleagues [Keller, et al. 80], is described here. It is of interest because despite the greater readability, there is still apparent a reasonably direct correspondence with the graphical form. Textual FGL has a syntax adopted from that of [Hearn 74].

For explanatory purposes, we shall use *upper case for literal tokens* and lower case to represent the names of syntactic entities. We use {...} to designate a sequence of *one or more* of the entity enclosed and [...] to designate *optional* syntactic entities. It follows that [{...}] denotes zero or more of the enclosed entity. We can then proceed with our definitions of program syntax by the following productions:

```
program -->                {block-definition}

block-definition -->       FUNCTION function-name [ argument-list ]

                           [ IMPORTS imports-list ]

                           [ LET abbreviation-list ]

                           RESULT result-expression

                           [ WHERE {block-definition} END ]

function-name -->          identifier

argument-list -->          identifier-list

imports-list -->           identifier-list

abbreviation-list -->      abbreviation [{ , abbreviation}]

abbreviation -->           identifier BE expression

result-expression -->      expression
```

An identifier list is defined as follows, where the symbol | denotes a choice of alternatives:

```
identifier-list -->          identifier
                           | ( identifier [ { , identifier } ] )
```

In FGL, an <u>identifier</u> is any sequence of letters, digits, or underscores (_) which begins with a letter or underscore.

One of the block definitions must have the function name <u>main</u>. It is this function which is evaluated by the system to cause the evaluation of all other functions.

As one can see, the only things that are not optional in a block definition are the function name and the result expression. In most cases, we will also have the first identifier list, which gives the names of <u>arguments</u> to the function being defined.

An <u>expression</u> is either a constant, an identifier, or one of the following:

```
    IF expression THEN expression ELSE expression

    n-ary-function (expression-list)

    unary-function expression

    nullary-function ()

    expression expression
where
    expression-list -->     expression [ { , expression } ]
```

Functions, either unary, nullary, or n-ary, can be either *atomic* or *programmer-defined*. An atomic function is one built into the language. A programmer defined function is what is being defined in a block.

As exceptions to the above syntax, some functions, e.g. binary arithmetic and logical, are represented in infix form.

An identifier used in a block definition must be *known* within the definition. There are five ways in which an identifier becomes known within a given block:

1. It is the function name.
2. It appears in the argument list.
3. It appears in the imports list.
4. It is defined in an abbreviation.
5. It is defined in the WHERE section of the block.

4.3.1. FGL has "block" structure.

The syntax rules impart a kind of "block structure" to textual FGL which is similar to the block structure of Algol, except that IMPORTS is used for making values known in an inner block, whereas in Algol these values are known implicitly. The nesting present in such block structure corresponds exactly to the nesting of *envelopes* which would occur if each block were treated as a function which is the argument to an apply wherever its name is used. In some respects, this nesting is similar to the "contour model" representation of an Algol-like program [Johnston 69, 71].

It is a matter of value judgment whether the explicit or implicit form of imports is preferred. The implicit form is more convenient when entering a program's text, but the explicit form is more useful when debugging a program. Given that the latter usually takes longer, we choose the explicit form.

Even with a compiler which recognizes common sub-expressions, it is occasionally tedious to write these sub-expressions multiple times in the code. For this reason, abbreviations are provided. The meaning of an abbreviation is that whenever the identifier occurs, it is equated with the expression. Notice that we do not preclude circularity in abbreviations. That is, A could be defined in terms of B, and vice versa. This is one way of textually representing the cyclic graph structures.

4.3.2. Example of textual FGL:

The 0-ary function which generates the stream of Fibonacci numbers could be coded as

```
    FUNCTION Fibonacci
    IMPORTS add_streams
    RESULT  cons(1, cons(1, add_streams(Fibonacci(), head Fibonacci())))
```

or alternately, using an abbreviation as

```
    FUNCTION  Fibonacci
    LET x BE cons(1, add_streams(x, Fibonacci))
    RESULT cons(1, x)
```

4.3.3. Another textual FGL example:

Here is how the serial_comp combinator (defined in Section 2.7.2) could be coded:

```
    FUNCTION serial_comp(f, g)
    RESULT  h
    WHERE
            FUNCTION h(x)
            IMPORTS(f, g)
            RESULT f(g(x))
    END
```

4.3.4. Example:

Another example is motivated by the presentation of a function REDUCE in [Iverson 79], which applies a binary function op to a non-empty list; i.e.

$$\mathrm{reduce(op)}(x_1, x_2, \ldots.., x_n) = \mathrm{op}(x_1, \mathrm{op}(x_2, \mathrm{op}(\ldots.., x_n)\ldots.))$$

The FGL version may be coded as

```
FUNCTION reduce(op)
RESULT  f
WHERE
        FUNCTION f(x)
        IMPORTS op
        RESULT  if null x
                    then nil()
                    else if null tail x
                            then    head x
                            else    op(head x, f(tail x))
END
```

5. Uses of the Graphical Formalism

5.1. Loop Removal

In contrasting the two odd-primes examples, one noteworthy point is that the first entails a graph with a loop (i.e. cycle) whereas the second does not. It is worth asking whether loops are in any sense necessary. In answer, we show that any loop can be removed, replacing it by an appropriate recursion. Hence loops are not essential, although there are implications which loops have on implementation which may make them useful for more efficient realizations. To wit, loops can be implemented as cyclic data structures and avoid a recursion. The following theorem illustrates the connection between loops and recursion.

> Loop Removal Theorem [Keller 77]: For every function graph, there is an acyclic graph (possibly with additional auxiliary nodes) representing the same function.

To prove the above, we first locate within the graph some cutset Y of arcs, i.e. a set of arcs, the removal of which makes the graph acyclic.

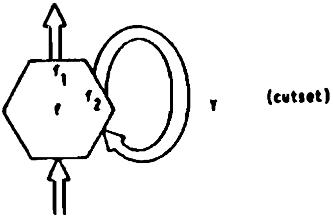


Figure 5-1: A graph showing the chosen cutset

Having chosen such a Y, depict the graph as in Figure 5-1. Here f represents the composite function of the acyclic portion of the network. f in turn is divided into f_1 representing the function which determines the values of the non-cutset arcs, and f_2, representing the function which determines the values of the cutset arcs.

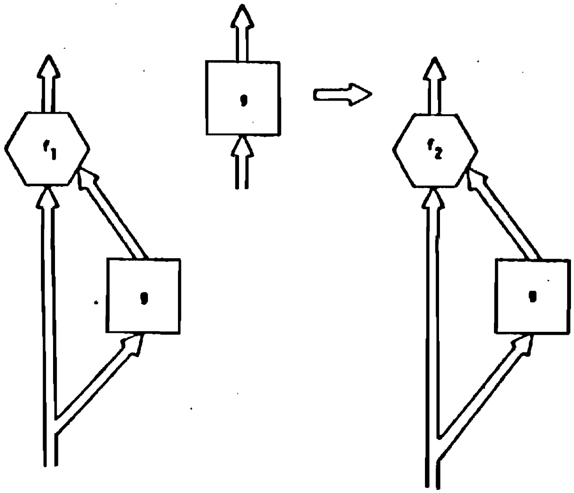


Figure 5-2: Acylic graph based upon the cutset in Figure 5-1.

We then introduce a new auxiliary, say g, and observe that the acyclic graph of Figure 5-2 is equivalent to the original, in that it has the same output function.

The validity of this construction is best explained by observing that the original network has

$$Z = f_1(X,\ Y)$$
$$Y = f_2(X,\ Y)$$

while the new network has

$$Z = f_1(X,\ g(X))$$
$$g(X) = f_2(X,\ g(X))$$

By identifying Y with g(X), the equivalence of the two is established.

If the cutset consists of fanned-out equivalents of the output arcs, then f_1 =

f_2. In this case, we can simplify the first equation to

$$Z = g(X)$$

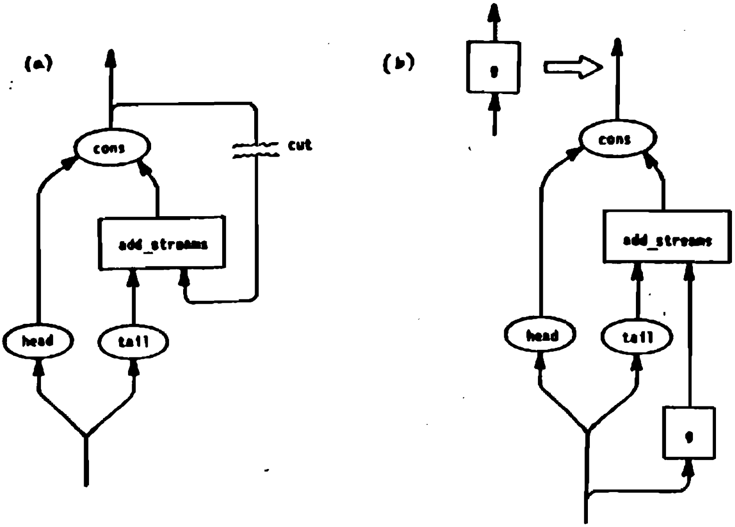


Figure 5-3: Example of cutset and simplified acyclic graph

Example: Figure 5-3 shows the choice of a cutset and result of loop removal in Figure 2-1. This illustrates the simplification mentioned above.

The converse of loop-removal is, of course, loop introduction. In the context of the execution model discussed earlier, loops make better use of storage than the corresponding recursive versions, as loops are not unrolled and do not require additional storage allocation. Techniques for loop introduction are still under investigation.

5.1.1. Tail recursion represents loops in conventional flowcharts.

There is a deceptive similarity between the loop removal theorem and a result appearing in [McCarthy 63a]. The latter demonstrates how any "flowchart program" can be converted into a recursive program. The idea is to replace iterations in the flowchart program with what are usually called "tail recursions". This technique is important, as it shows that any flowchart program has an equivalent representation in our graph formalism.

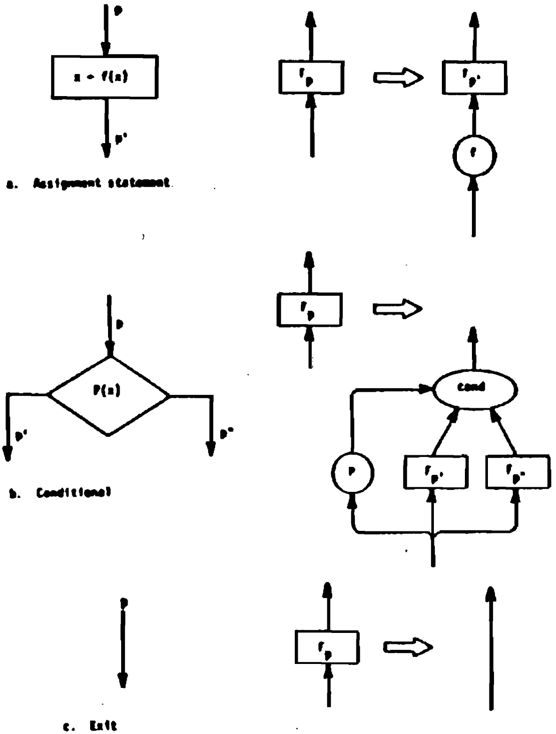
> Tail Recursion Theorem [McCarthy 63a]: For any flowchart program, there is an equivalent graph grammar, in the sense that there is an auxiliary node in the latter which computes the same function as the program.

We attempt to convey the basic idea of the proof without entering into a formal presentation of what is meant by a flowchart program. Such a program consists of statements which operate on "program variables". Let x be the vector of such variables. We create from our program a set of productions, the underlying functions of which operate on the product data type of values which x may assume.

For each "control point" p in the flowchart, we introduce an auxiliary symbol F_p. The idea is that $F_p(x)$ represents the transformation undergone by x if the program is started at point p. It will turn out that the function F_1 corresponding to the initial (i.e. entry) control point is the function computed by the flowchart, for arbitrary initial program variable values.

It suffices to demonstrate what productions are introduced for each flowchart box. Here we treat only assignment statement boxes and test boxes. An assignment, in full generality, appears as in Figure 5-4a. The corresponding graph grammar production is also shown there. A test appears as in Figure 5-4b, with its corresponding production. Here we have used a terminal function cond, defined earlier. Finally, the production of Figure 5-4c is introduced for each exit point.

Of course, simplifications are possible. For instance, by composing functions in a fairly obvious way, we only need one auxiliary for each loop in the



a. Assignment statement

b. Conditional

c. Exit

Figure 5-4: Productions equivalent to flowchart constructs

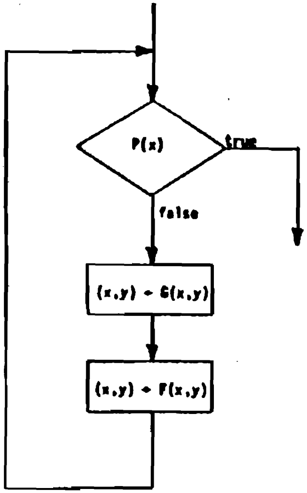


Figure 5-5: Factorial Flowchart

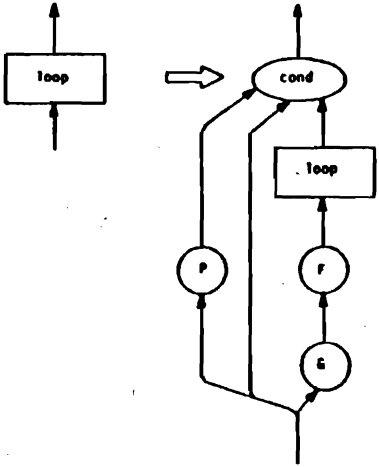


Figure 5-6: Factorial Production

flowchart. Also, recognizing that the functions in the flowchart don't usually operate on <u>all</u> of the program variables, optimization is possible which produces more "independent" arcs (see Section 5.3).

5.1.2. Example:

Applying the tail recursion transformation to Figure 5-5, we have the equivalent production in Figure 5-6. We call the programs in these two figures "Factorial", since if we interpret P, F, and G as

$$P(x): x \leq 1$$

$$F(x, y): (x - 1, y)$$

$$G(x, y): (x, x * y)$$

and if y is initially 1, then we have programs for computing x Factorial.

It has been shown in [Paterson and Hewitt 70] that, barring the addition of new functions to the program, the reverse transformation (from a production form to a flowchart) is not generally possible. A consequence of the following section is that the transformation is possible if we are allowed to use additional functions.

5.2. Production Removal and Explication of "Paradoxical" Combinators

Our graphical formalism possesses several potentially pedagogical uses, including the ability to understand "paradoxical" combinators, or "Y" operators. These are various arcane ways of achieving the effect of recursion without the explicit use thereof.

One method of removing recursion is to extend the function enveloping device described in the Section 2.5. Suppose that we have a recursive production of the form of Figure 2-19. There we used the symbol H to denote the graph which is the consequent of the production. We stated earlier that H could be re-expressed as H' in Figure 2-22, where we have replaced the occurrence of G inside H with an apply operator.

But since G is supposed to be replaceable by H, we have another folded version

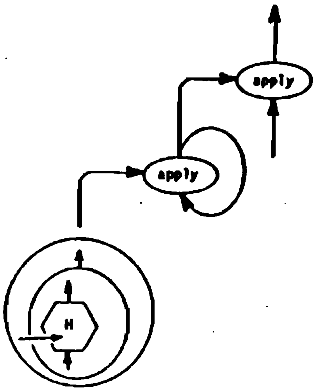


Figure 5-7: Folded version of Figure 3-11b

as shown in Figure 5-7, which is also free of the auxiliary G. To reassure ourselves of the equivalence of this graph and the G in Figure 3-11b with which we started, use the apply rule of Figure 2-16.

5.2.1. All loops and productions can be removed.

We now investigate the possibility of eliminating all loops and productions. The preceding discussion shows how to get rid of productions. The resulting graph, of the form shown in Figure 5-7, has a loop. But if this loop can be eliminated without introducing other loops or productions, then we shall have found a way to eliminate all loops and productions, since this loop is used to achieve the effect of a prototypical production. Experience suggests a way of achieving this goal.

Consider the subgraph Y of Figure 5-7, as shown in Figure 5-8. We notice that

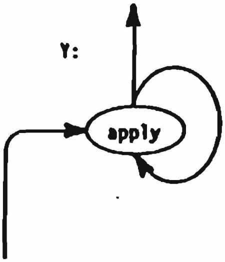


Figure 5-8: Subgraph Y of Figure 5-7

if F is present on the input arc of Y, then the output arc z must have the property

$$z = F(z)$$

In other words, the function Y represented by the graph is such that for any function F, Y(F) is a function such that

$$Y(F) = F(Y(F))$$

That is, Y produces a *fixed point* of F. The above only makes sense, of course, if F is a function-producing function, e.g. M' in the current example. Since $Y(M') = M'(Y(M'))$, we used repeated substitution to observe

$$Y(M') = M'(Y(M')) = M'(M'(Y(M'))) = M'(M'(M'(Y(M'))))$$

the limit of which gives us

$$Y(M') = M'(M'(M'(M'(...))))$$

which is exactly what we get by repeated substitution of M for G when expanding productions.

Fortunately, a loop-free operator Y' equivalent to Y is known, and is shown in Figure 5-9.

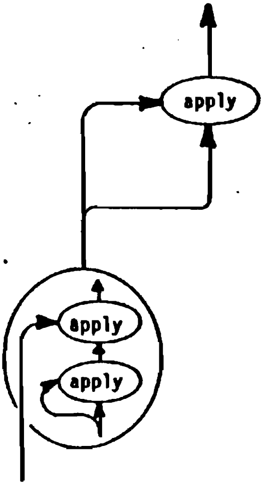


Figure 5-9: Loop-free operator Y' equivalent to Y

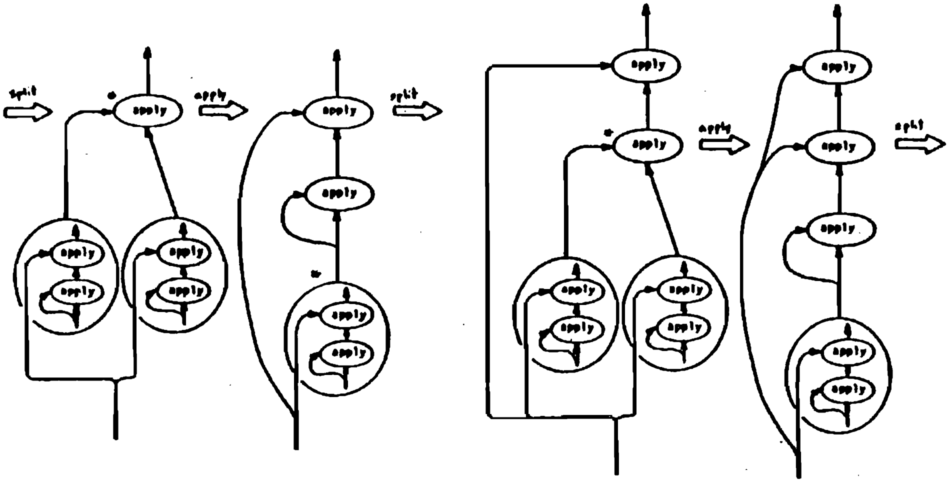


Figure 5-10: Showing the equivalence of Y and Y'

We present in Figure 5-10 an enlightening graphical argument to show the equivalence of Y' and Y. The limit of the sequence in Figure 5-10, if an enveloped H' is provided as an argument, is the same as that in the subgraph of Figure 3-11b, which represents the least fixed point of H'.

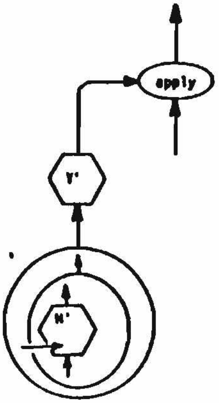


Figure 5-11: Application of the operator Y'.

In summary, Y' is a loop-free equivalent of Y, which allows G, defined by an equivalent loop-free graph, as shown in Figure 5-10. Readers familiar with the lambda calculus [Church 41] will recognize the lambda calculus expression of Y' as

$$\lambda f.((\lambda g.gg)(\lambda x.f(xx)))$$

although we feel that the graphical version is much clearer. Since lambda calculus expressions are essentially loop-free, there appears to be no direct way to represent the Y in Figure 5-8.

In summary, we have used a graphical technique to demonstrate why the "paradoxical combinator" [Curry and Feys 58] is usable to attain the least

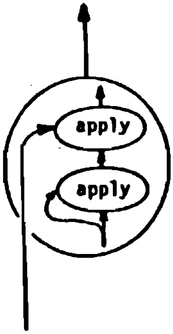


Figure 5-12: A distinguished sub-structure of Figure 5-9

fixed point $H'^{\infty}(?)$. In retrospect, we see that the sub-graph of Y' shown in Figure 5-12 which appears at each stage in the expansion of Y', is somewhat arbitrary. It is not present at all in the limit graph. Indeed, its use appears to be mainly to force the infinite expansion. With this in mind, it is probably no surprise that there are many other such structures which will suffice for this purpose, with no two being inter-convertible by simple transformations such as split and apply. We offer this as a reason for the existence of such functions, as mentioned for example in [Wadsworth 76].

5.3. Parallelism

One important use of function graphs is in the exhibition of opportunities for concurrent (or parallel) evaluation. Parallelism shows up naturally in a function graph in the form of two or more independent arcs, i.e. arcs not lying on a common chain of arcs. The nodes which have those arcs as their outputs indicate functions whose computation can proceed concurrently.

In the sub-graph of Figure 5-13 for example, arcs x, y, and z may receive values concurrently.

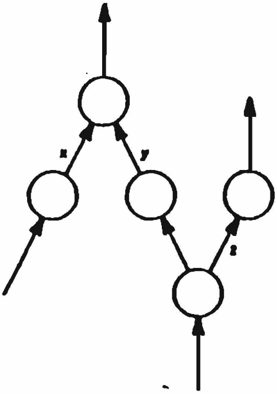


Figure 5-13: Independent arcs which are evaluable concurrently

Graph grammar productions may be used for generating computations with arbitrary amounts of parallelism, depending on the input data. Consider, for example, the problem of computing the number of leaves of a tree data structure. We can initiate arbitrarily-many sub-computations which compute the number of leaves of selected sub-trees, then proceed to add the resulting values.

The production of Figure 5-14 defines such a function. We show in Figure 5-15 the result of the partial evaluation of the function after it has been applied to a tree having 256 leaves.

5.3.1. Parallelism occurs in different granularities.

We would conjecture that most ways of exploiting parallelism in programs are all instances of this "independent arc" phenomenon. For example, the processing unit of a "look-ahead" processor (cf. [Keller 75]) dynamicaly constructs such a graph from a sequential program to determine concurrently-executable functions. Although it is tempting to differentiate between "look-ahead", "pipelining", and other forms of parallelism, such differences are essentially a matter of the *granularity* of the parallelism rather than being distinct conceptually.

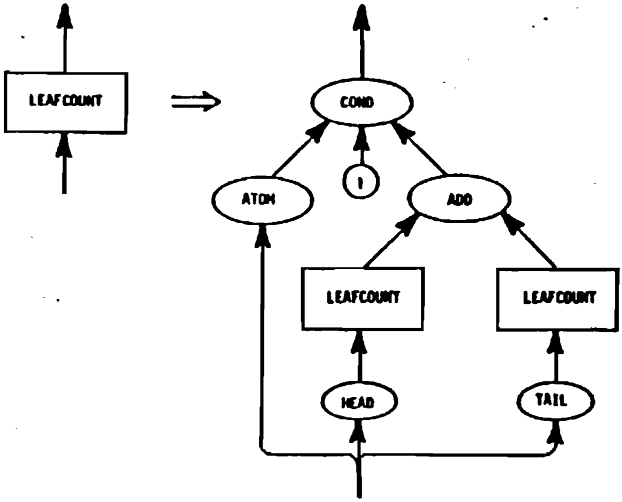


Figure 5-14: Production for the leafcount function

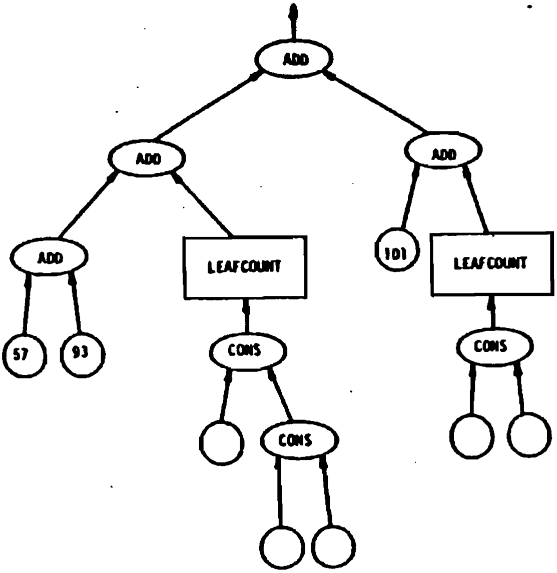


Figure 5-15: Leafcount evaluation

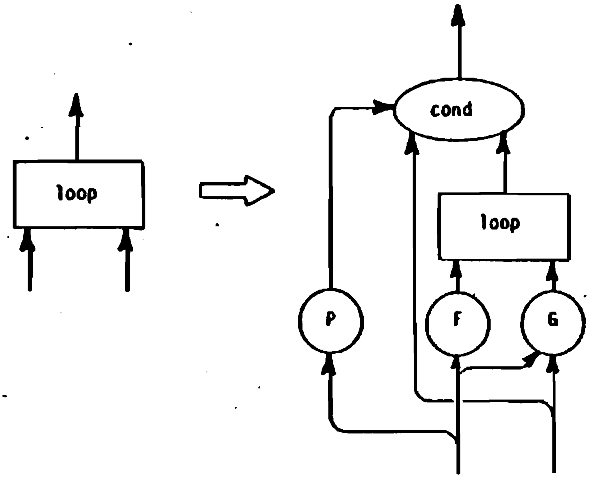


Figure 5-16: Production transformed from that of Figure 5-6

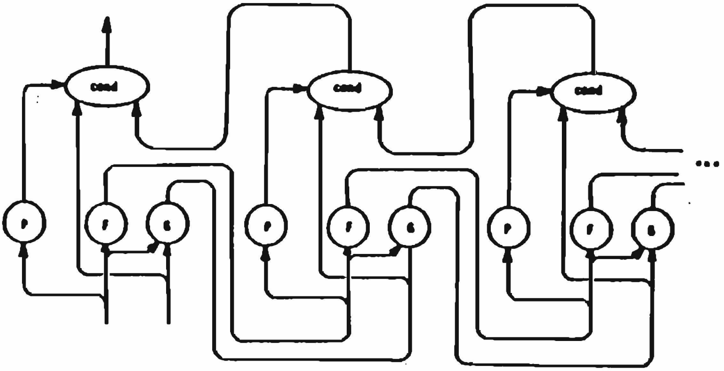


Figure 5-17: Unwound graph corresponding to the production of Figure 5-6

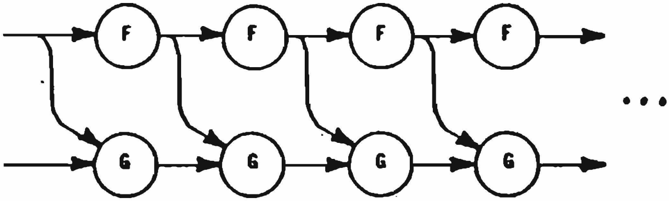


Figure 5-18: An instance of the graph in Figure 5-17 with Ps evaluated

We now illustrate in the function graph model how this look-ahead phenomenon occurs in computing operations from several different iterations of a loop. Recall the flowchart of Figure 5-5 which was transformed to the recursive production in Figura 5-6. In Figure 5-16, we have further transformed the production by separating the variables to make independent are parallelism more evident. In principle, this production represents the infinite graph shown in Figure 5-17.

For sake of clarification, suppose the first several Ps in this graph evaluate to 'all' (false). We then effectively have the graph of Figure 5-18. We see from the above that any i^{th} instance of G, counting from left to right, can be executed concurrently with any j^{th} instance of F, as long as $i \leq j$. A similar fact was used in [Keller 73] to show that no finite amount of control storage generally suffices to achieve maximal parallelism. In this example, we see that no finite amount of intermediate data storage suffices either, since if G is much slower than F, arbitrarily many of the intermediate results of different Fs must be saved to compute future Gs.

5.3.2. Auxiliary nodes can temporarily mask parallelism.

The graph model seems to be capable of displaying much of the parallelism inherent in a program. One caution should be taken, however, in assigning arcs to be dependent simply because they lie on a common chain. This caution amounts to the fact that arcs entering and leaving a node are not necessarily connected in principle.

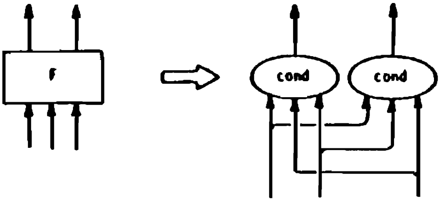


Figure 5-19: The antecedent and consequent graphs are equivalent only when evaluated with an appropriate evaluation rule

For example, suppose we have a node f which is defined by the equation

$$f(x, y, z) = (\text{if } x \text{ then } z \text{ else } y, \text{ if } x \text{ then } y \text{ else } z)$$

This function might be represented by the production of Figure 5-19.

However, whether this production is an accurate representation of the equation depends heavily on the substitution mechanism used in effecting productions. If the mechanism uses the demand-driven scheme suggested in Section 4.2 to effect the replacement suggested, there is no difference. However, some methods of evaluation, variously known as "data-driven" or "call-by-value" would require data to be present on *all three* arcs in order for the expansion to occur. The picture represented in the production is then not accurate. Instead, we have a function f' defined by

$$f'(x,y,z) = \begin{cases} f(x,\ y,\ z) \text{ if } x \neq ?,\ y \neq ?, \text{ and } z \neq ? \\ ? \quad \text{otherwise} \end{cases}$$

which is clearly undefined on some arguments where f is defined.

Put another way, f' has a *synchronizing* effect in having to wait on all of its values, whereas f does not. Synchronization is contrary to parallelism, since it introduces extra dependencies between operations.

The same phenomenon is observable in the choice of our definition of the operator cons in the Evaluation section. Our cons follows the spirit of [Friedman and Wise 76] and [Henderson and Morris 76] in being a cons which "does not evaluate its arguments", or one which is implemented by "lazy evaluation". More precisely, the equations

$$\text{cons}(x,y) = (x,y)$$

$$\text{head}(\text{cons}(x,y)) = x$$

$$\text{tail}(\text{cons}(x,y)) = y$$

$$\text{select}(i,\ \text{cons}(x_1,\ \ldots,\ x_n)) = x_i$$

all hold without qualification on x and y. In contrast, cons in conventional Lisp and languages designed for data-driven execution is *strict*, i.e. requires all arguments to be "complete" prior to yielding any result, thus producing a strong form of synchronization. By a *complete* argument, we mean one which is a finite tree with no undefined leaves. Our cons is *lenient*, in that it does not require any argument to be complete to yield a meaningful result. Lenient cons provides no synchronization at all, but simply has the effect of making a value from a tuple of values. This value can be treated as a single entity, later to be decomposed by select functions.

5.3.3. Lenient operators simplify understanding and proofs.

Another feature of the lenient form of operator is that for ascertaining the correctness of a program, we wish to be concerned as little as possible with stipulations such as "if x ≠ ?". With lenient operators, there are no such

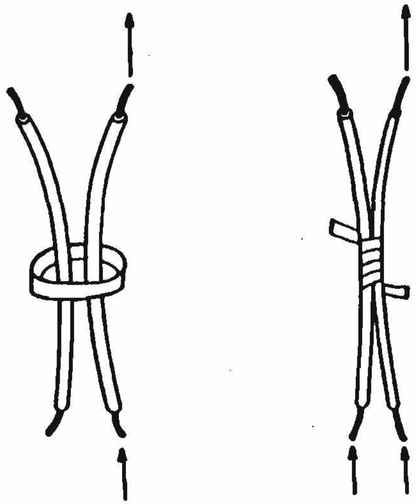


Figure 5-20: Wiring analogies to the cons operator: In the left pair, corresponding to lenient cons, the component wires are paired, and either wire can be pulled without pulling on the other. In the right pair, corresponding to strict cons, the wires are bound, and pulling either wire effectively pulls both.

Figure 5-20 illustrates the difference between lenient and non-lenient cons through a wiring analogy. If we view the arcs on which values flow as wires, then in the non-lenient version, the two wires are wrapped together. Pulling on either output wire pulls both of the input wires, and the output wire doesn't respond unless both input wires are free. In the lenient version, pulling on an output wire pulls the corresponding input wire, independent of the other input wire's connection.

5.3.4. Data type ordering affects degree of concurrency.

The greater asynchrony, and hence concurrency, available with lenient cons manifests itself in another way. It allows us to use the tree ordering for our data type (see Section 3.2), as opposed to a flat ordering. The tree ordering implies a finer grain of observable step in the production of data objects than does the flat ordering of the same objects. With the flat ordering, there is an all-or-nothing behavior of each function, i.e. the only allowable progression of a value is from totally undefined to completely defined in one step. In contrast, the tree ordering allows an infinity of gradations, including the possibility of an infinite series of approximations, none of which ever arrives at a completely defined object, but each of which is itself useful. Hence the data type ordering serves as a valuable indicator of the granularity and hence the degree of attainable concurrency.

5.3.5. Lenient cons enhances asynchrony.

We mention that lenient cons automatically includes the capability of achieving greater asynchrony in stream-oriented computations than do stream operators which are restricted to process stream items in strict order. This asynchrony in turn lessens the constraints on the computation, thereby producing more opportunities for concurrent evaluation. Such differences in modes of interpretation have been observed, for example, in [Arvind and Gostelow 78] which mentions an "unraveling interpreter". Such an interpreter is, in fact, implied by a language semantics which provides a lenient, rather than strict, cons operator.

There is a pleasing connection between the lenient cons convention and the splicing effect of graph grammar productions. It indicates that we may restrict our attention to auxiliary nodes with only one input and one output arc, since any number of arcs may be coded and decoded using cons and select. As long as lenient cons is used, the effect is the same. The diagram of Figure 5-21 illustrates.

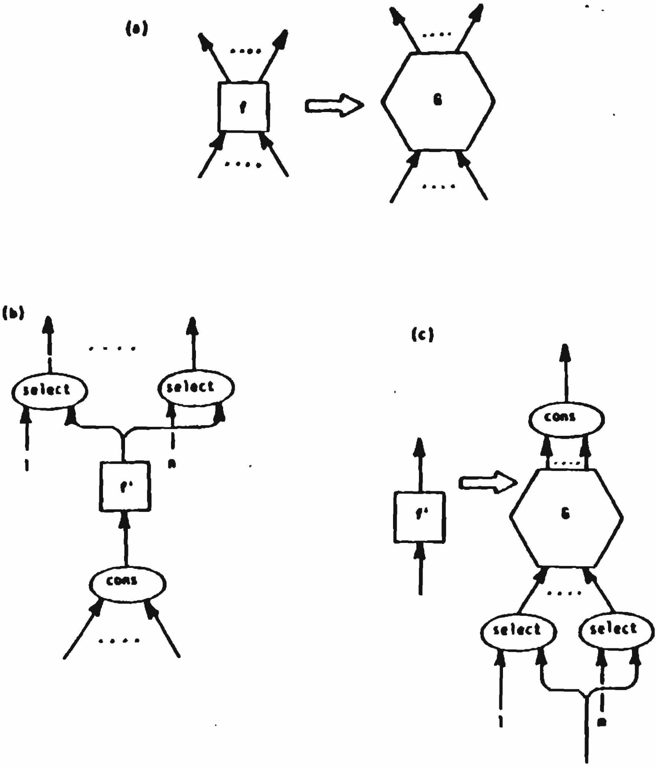


Figure 5-21: Replacing auxiliaries with auxiliaries having only one input and one output arc: (a) Original production; (b) Replacement for antecedent; (c) New production

5.3.6. Transparent functions allow programmer control of concurrency.

The final discussion of this section concerns the exertion of greater control over the amount of concurrency actually realized in evaluation. Let us assume the demand-driven scheme discussed earlier. Assume further that we have coded the productions for some computation. A property of the demand-driven scheme is that it will never begin evaluating some object until it has determined that the object is actually needed. However, the programmer may well know that certain objects are ultimately going to be needed prior to their need being perceived by the evaluator. To allow these needs to be injected as additional demands, we can provide a special operator, par. This will be a generic operator with any number of arguments. Its definition allows it to be rather transparent functionally:

$$par(x_1, x_2, \ldots, x_n) = x_1$$

However, its effect on the demand evaluator will be to propagate demand to all of its arguments immediately. This will have the effect of anticipating the need for those arguments and forcing them to be recognized as concurrently evaluable.

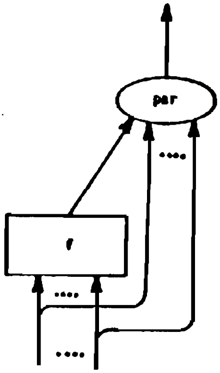


Figure 5-22: Use of par

A typical use of par, to evaluate the arguments for an auxiliary node concurrent with its expansion, is shown in Figure 5-22.

A dual problem involves an observed "time-space tradeoff". It takes memory space to support concurrent activities, directly proportional to the number of such activities. It might therefore be desirable to have an operator which reduces concurrency, thereby reducing memory requirements. This can be done by intentionally sequencing the evaluation of operations which could otherwise be evaluated concurrently. The definition of such an operator is

$$\mathrm{seq}(x_1,\ x_2,\ \ldots,\ x_n) = \begin{cases} x_n & \text{if } x_1 \neq ?,\ x_2 \neq ?, \ldots,\ x_n \neq ? \\ ? & \text{otherwise} \end{cases}$$

For the demand evaluator, seq demands each argument in turn, somewhat like our cond was specified to do. Only when all demands have been satisfied does it return its last value. A use of seq, depicted in Figure 5-23, is to prevent expansion of a production until all of its arguments are ready.

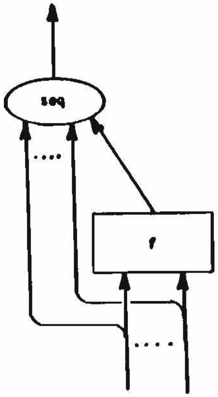


Figure 5-23: Use of seq

Neither seq nor par is appropriate in some situations. For example, if one wanted all arguments to be ready before expanding a production, but wanted the arguments to be evaluated concurrently, then an operator such as apar (for strict par) could be used. The functional definition of spar is similar to that of seq, but demands propagate to all arguments concurrently.

Other similarly useful operators are under current investigation. A clean style of programming seems to result from initially using lenient operators as much as possible, then "overlaying" on the program operators such as seq and par for greater control. The varieties of such operators seem to point to a need for an operational semantics to "overlay" the denotational semantics of functional languages. This remains a topic for future investigation.

5.3.7. Variations on operators effect demand-driven execution.

It is worth noting the similarity between par and operators such as the <u>parallel</u> <u>conditional</u> and <u>parallel</u> <u>or</u> [Kleene 52], [Paterson and Hewitt 70]. For brevity, we discuss only the first of these.

Our cond operator has been defined in Section 4.2. It is possible to devise a different operator, pcond, which has an effect similar to par in that it demands all of its arguments, plus an additional aspect which gives defined results in some cases where cond does not. The definition is

$$\text{pcond}(x,y,z) = \begin{cases} y & \text{if } x \neq \text{'nil'} \\ z & \text{if } x = \text{'nil'} \\ y & \text{if } y \underline{=} z \\ ? & \text{if } x = \text{?} \text{ and not } y \underline{=} z \end{cases}$$

Here $\underline{=}$ is some "weak equality" predicate. That is, it does not test true equality of its arguments, but rather some weaker relationship which implies equality, such as being the same storage structure.

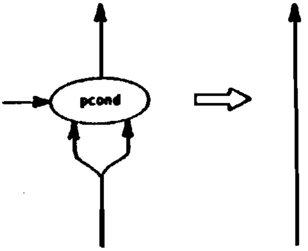


Figure 5-24: A reduction possible for pcond but not for cond

Such a form of equality takes place, for example, in the graph reduction rule shown in Figure 5-24. The reason why true equality is not elected for $\underline{=}$ is that the former would generally be an uncomputable predicate.

As mentioned, poond will give more information (in the sense of the data type ordering) than cond. It appears, however, that this benefit will be reaped only rarely in practice. Since poond requires propagation to all arguments, yet will often be unable to make use of the value of one of them, it seems that poond will generate more work than it saves, unless a superfluity of otherwise idle processors is available. The use of poond-like operators for gaining parallelism is discussed in [Friedman and Wise 78].

To summarize this author's opinion, it is generally more efficient to rely on strict operators to introduce concurrent demands for values known to be essential than to use poond-like operators which will only yield benefits in a small number of cases.

5.4. Ancillary Applications

We mention in this section an additional application of the function graph concept, where by "applications" we mean other models which may be viewed as instances of function graphs. These applications fall within the realm of the "general" theory, in that they do not have a direct correspondence with an execution model. The intended result of such a pursuit is that methods being developed for proving properties of function graphs are then applicable to these applications.

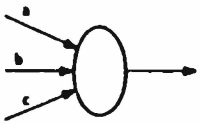


Figure 5-25: Function of a node of a graph operating on languages

5.4.1. Language theory uses function-graph ideas.

One application is to formal languages (i.e. sets of strings over a finite alphabet). In the language context, nodes of a function graph are viewed as functions on languages. Specifically, each node with n input arcs is the union of n languages, each formed by concatenating to each member of the input language the string which labels the arc. Thus, the function of the node shown in Figure 5-25 we have

$$f(L_x, L_y, L_z) = L_x a \cup L_y b \cup L_z c$$

It has long been understood that finite-state languages can be represented by labelled directed graphs without use of productions, or equivalently, by "regular expressions" [Kleene 56]. Similarly, context-free languages can be represented by a kind of graph grammar called a syntax graph (cf. [Reeker 71], [Hoare and Wirth 73]). We simply wish to point out that such representations can be viewed as function graphs if properly interpreted, and that the corresponding interpretation of the determinacy theorem is that in which the least fixed-points are just the languages generated.

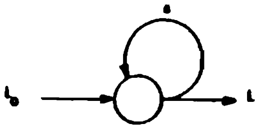


Figure 5-26: A function graph representable by a regular expression

Consider the labelled graph representation of the non-deterministic finite-state machine in Figure 5-26. To see the connection with languages, notice that in the interpretation of the node state above, we have an equation such as

$$L = G(L)$$

where

$$G(L) = L_0 \cup La.$$

But we have already represented the solution of this equation as

$$G^*(\emptyset) = L_0 \cup (L_0 \cup (L_0 \cup \ldots.)a)a)a$$

which is usually denoted

$$L_0 a^*$$

in the notation of regular expressions. By such reasoning, we see that the function described by any graph is representable by a regular expression, a result attributed to [Kleene 56].

If we allow productions in addition to the type of operator shown in Figure 5-25, it is known [Reeker 71] that the function is not generally representable by a regular expression, but in fact requires the greater power of a syntax graph (equivalently, a context-free grammar).

5.5. Indeterminacy

We have observed the Determinacy Theorem for function graphs, which states that each graph determines unique output values on each of its arcs, given particular initial values on its input arcs. However, it is known that there are perfectly reasonable computational systems which do not enjoy such determinacy properties. An often cited example is that of an airline reservation system, wherein the net result, the set of passengers departing of a given flight, might well be dependent on internal system timings, even given a fixed set of requests for seats.

The difference between indeterminacy and "non-determinism" should be mentioned. Non-determinism refers to the system choosing one of several actions in a manner "local" to the behavior of the system. Such behavior might well be prevalent in all of the systems discussed in this paper. On the other hand, indeterminacy is a global phenomenon which says that the overall outcome of a system's execution may be one of several or many possibilities.

Although such indeterminate systems have been the result of some study (cf. [Plotkin 76], [Smyth 78], [Keller 78a], [Kosinski 79]), no satisfactory general theory has been developed analogous to the one presented so far. The operational (i.e. state-transition) description of indeterminate operators is usually fairly simple, yet attempts at describing them as functions over their input histories have been unsuccessful.

As an example, consider the merge operator. It operates on two incoming streams of values and produces a stream which is a shuffle of the two input streams. A given pair of input streams may well have many different shuffles, e.g. the sequence a b shuffled with c d yields

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

and

c d a b

Here then is an example of indeterminacy.

5.5.1. Indeterminate operators challenge conventional Intuition.

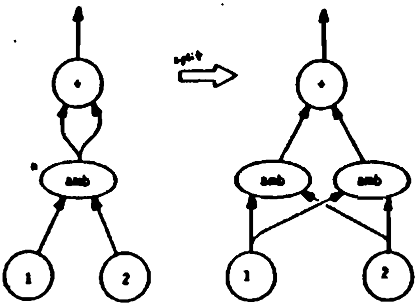


Figure 5-27: Splitting involving amb

Reinterpreted here in the context of function graphs, [Hennessy and Ashcroft 77] illustrate that referential transparency, as exemplified by the splitting rule, is destroyed when indeterminate operators are allowed. For example, consider an operator amb [McCarthy 63b] which is defined by

$$\text{amb}(x,y) = \begin{cases} x \text{ if } x \neq ? \\ y \text{ if } y \neq ? \end{cases}$$

with amb(x,y) being indeterminately x or y if both expressions hold. To see where referential transparency fails, consider the expression

$$\text{amb}(1,2) + \text{amb}(1,2)$$

Depending on whether the left or the right graph in Figure 5-27 is used, the possible results could either be in the set {2, 4} or in the set {2, 3, 4}. [Ward 74] considers additional ramifications of amb-like functions. Similarly, [Giordano 79] shows that the cycle removal result described in Section 5.1 does not work with indeterminate operators.

[Keller 78a] observed that graphs which incorporate merges can exhibit anomalies when we attempt to define a functional semantics for merge. For example, when a merge occurs in a cycle, it is possible to exhibit shuffles of its ultimate inputs which are impossible as ultimate outputs. From this argument, one can further conclude that a merge cannot be described as a function on any product of any two data types. Instead, a functional description of merge must take into account relationships between items in the two streams which occur because of timing within the system.

5.5.2. Some indeterminacies are benign.

A class of systems intermediate between two extremes is that in which there are local indeterminacies in streams of output values, yet a unique ultimate value is always produced on outputs of interest. A paradigm for handling such cases is to attempt to modify the ordering part of some of the data types in such a way that the operators become continuous functions with respect to the modified ordering, then apply the determinacy theorem. When the determinacy theorem can be applied, the verification of the system can be reduced to the verification of selected sequential executions. An example of this approach appears in [Keller 78b].

5.6. Proof Methods

Most existing methods of correctness proving can be couched in terms of proofs for function graphs. Likewise, for most known proof methods applicable to function graphs, there are also known instances of such methods which have been applied to more specific models. The one thing we feel is to be gained in attempting to express a program graphically is that the latter viewpoint may suggest additional avenues of attack for proofs. In this section, we present proof methods specifically from the graph viewpoint.

5.6.1. Inductive proofs come in several interrelated forms.

Any general proof method which deals successfully with infinite objects, e.g. general data types and functions represented by graph grammars, is going to use some form of induction. There are several ostensibly different forms of induction, namely:

1. Induction on the data objects which are arguments to functions.
2. Induction on the structure of the program.
3. Induction on the sequence of steps taken in execution of the program.

Despite these apparent differences, the forms of induction are often closely related and sometimes the difference is only one of viewpoint.

For example, the class of data objects of interest is often representable using a (perhaps non-deterministic) production which generates the class. So induction on the structure of a generating program might be used to get the same effect as induction on data. Similarly, if we are allowed to treat our graphs as data, as has already been done to some extent in the discussion of enveloping, then we might well find that induction on programs is essentially induction on data objects representing programs. Finally, we can often model the execution sequences of n program as a data-type in another closely-related program, so that induction on execution sequences may also be turned into induction on data.

The scope of this paper does not permit an exhaustive survey of inductive methods and their classification. Instead, we must be content with a few examples of how inductive proofs can be performed in the function graph context.

5.6.2. Information and proof orderings may differ.

Let us begin with a discussion of induction on data. We have already mentioned that the notion of a data type includes an ordering on the members of its domain. To do induction, we also need an ordering, but the two orderings need not coincide. More stringently, the type of ordering needed for data induction must be an inductive ordering, i.e. a partial ordering < with no infinite descending chain,

$$x_0 > x_1 > x_2 \dots$$

This property is necessary because of the way in which induction proceeds, i.e. by means of a basis and an induction step.

If we are attempting to prove a property P for all members of a data type, then in the basis we prove P(x) for all minimal elements x, where by minimal we mean that there is no y such that y < x. Such elements must exist, because if we start with an arbitrary element and repeatedly choose "smaller" elements, forming a descending chain, then the chain cannot descend forever (due to the ordering being inductive) and therefore must stop at a minimal element.

In the induction step, we assume that x is an arbitrary non-minimal element of the data type. We show that

If for each y < x we have P(y),

then also P(x)

Here the first line is called the inductive hypothesis and the second is the inductive conclusion. This particular version of the induction step actually embodies the basis as well, in that for a minimal element x, we must prove P(x) directly. We separate the basis of the proof from the induction step in order to decompose the discussion by treating only non-minimal elements x in the induction step. Once the basis and induction step are shown, the conclusion is that P(x) holds for every possible x in the domain.

The catch in this form of proof is the generality of the induction step. It must work for <u>all</u> non-minimal x. The ease with which this may be proved governs the choice of the inductive ordering, which may be quite unlike the information ordering of the data type.

5.6.3. Example of Data Induction:

Consider the function sum_stream defined in Figure 2-1. Suppose that the input stream to the function is

$$x_1, x_2, x_3, \ldots.$$

We want to show that the output stream

$$y1, y2, y3, \ldots.$$

has the property

$$y_j = x_1 + x_2 + x_3 + \ldots. + x_j, \text{ for each } j \leq i$$

Here we can use data induction, choosing our inductive ordering as the prefix ordering on streams.

As a <u>basis</u>, we suppose that x is the null stream. The property clearly holds in this case, as the output is also the null stream, according to the definition of sum_stream.

As the <u>inductive step</u>, suppose that x is not the null stream and the property holds for all z which are proper prefixes of x. In particular, x has some non-zero number of components, say i, and the property holds for the prefix of length i - 1. More precisely,

$$y_{i-1} = x_1 + x_2 + x_3 + \ldots. + x_{i-1}$$

The $i-1^{th}$ component of the stream output of the add_streams will therefore be $y_{i-1} + x_i$. But this component is also the i^{th} component of the output y. Hence

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Combining the above with the inductive hypothesis, we have the inductive conclusion.

The inductive proof method above is only partially complete, as we have assumed thus far that the input stream x is *finite*. To make it complete, we must observe that the truth of the conclusion for an *infinite* x follows from its truth for all its finite prefixes. In this example, the observation indeed holds. To see why, suppose that the statement is true for all finite x, but there is an infinite x such that for some i it is not true that

$$y_i = x_1 + x_2 + x_3 + ... + x_i$$

Then clearly the conclusion must also fail when the finite prefix of length i of y is the input, which is a contradiction.

5.6.4. Admissibility makes proofs work for infinite objects.

The quality of a predicate P, that the truth of P on infinite objects follows from its truth on smaller finite truncations, is called *admissibility*. It is a special case of the concept of continuity of functions on data types as discussed earlier. In particular, if we view a predicate as a function into the data type with domain {true, false} which has the ordering true < false, then continuity with respect to this type is the same as admissibility.

It is easy to construct examples of predicates which are not admissible in the above sense. Consider, for example, the natural numbers, with infinity added, and the numeric ordering. Let P(x) be "x is finite". Then the basis P(0) holds and the inductive step holds for finite x. Also the inductive hypothesis holds for infinity. However, the inductive conclusion most certainly does *not* hold for infinity. Hence this P is not admissible.

5.6.5. Proofs for sequential programs can be cast as function graph proofs.

We now discuss *induction on execution sequences*. In particular, we discuss execution of flowchart programs, and show how interpret this form of induction as induction on data in the function graph model. A typical and widely used version of induction on execution sequences is one used to prove that an assertion about the values of program variables holds when the program terminates, assuming that another assertion about the values of those variables held when the program started. This method is widely attributed to [Floyd 67], although its essence appeared in [Gorn 59]. To apply the method, it is often necessary to add other assertions about the values of variables at other points in the program.

To view the above method in the function graph model, we think of each flowchart statement as a function on the set of sets of program variable states. For example, if the variables are (x, y, z), then the arc data type is the set of all sets of values which can be assumed by the triple (x, y, z). Each statement corresponds to a function on this set of sets. For example, corresponding to the statement

$$x := y + z$$

we have the function F given by

$$F(S) = \{(x', y, z) \mid (x, y, z) \text{ in } S, x' = y + z\}$$

A similar viewpoint can be used to see that conventional "flowchart programs" are just special types of function graphs. In this case, the data type is that of *sets of state vectors*, i.e. vectors of values assigned to variables of the program. The statement nodes of such a program are just functions on these sets. For example, an assignment statement

$$x := F(x)$$

when viewed this way is a function G defined by

$$G(S) = \{F(x) \mid x \text{ in } S\}$$

for any set S of state vectors.

Similarly, the merge of two flowchart arrows is the union of the two sets of state vectors. An equivalent viewpoint is that of *predicate transformers* [Dijkstra 76], since a predicate in such a program is the same as a set of state vectors.

[Hoare 69] introduced a method for axiomatizing the introduction of assertions. He indicated how axioms could be presented which generate atomic statements accompanied by assertions and how rules of inference could be used to generate compound statements accompanied by assertions. This method could therefore be considered *induction on program structure*. We wish to indicate that a similar approach can be used for function graphs. This approach is a generalization of Hoare's in that it can be applied to data types other than sets of sets of program states.

5.6.6. Assertional proof methods extend to function graphs.

We may accompany any function graph with an assertion about its input/output relation. This might be decomposed into an implication which involves a hypothesis about the ingoing value and a conclusion about the outgoing value, but other forms of assertions are possible. For atomic functions, the allowable assertions are derived *ad hoc* from the semantics of those functions. For non-atomic functions, composition rules must be developed which derive the assertion for the function from the assertions for its constituents.

An example for a series interconnection of two graphs is shown in Figure 5-28. In the case that the assertions are decomposed into the type of implication mentioned above, we have a simpler composition rule, as shown in Figure 5-29. Similarly, when one operator is a cond, we may use the rule in Figure 5-30.

5.6.7. Fixed point induction proves properties of functions.

A very important rule is the *fixed point induction* rule, which gives us a way of proving properties of recursively defined functions. The rule is shown in Figure 5-31. Recalling that $M^*(G)$ is the function computed by the recursive

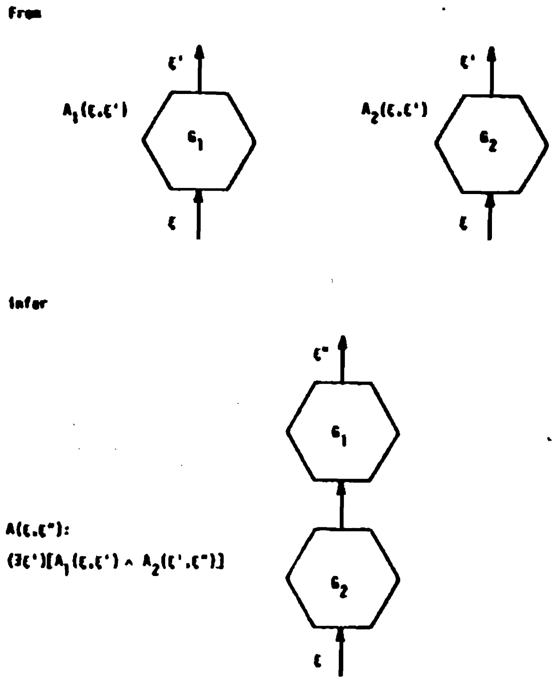


Figure 5-28: Composition rule for a series interconnection

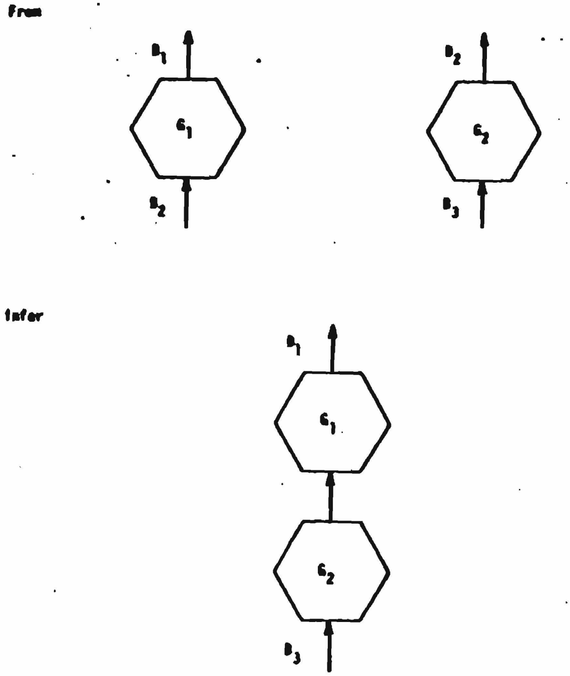


Figure 5-29: Special case of the composition rule for series interconnection

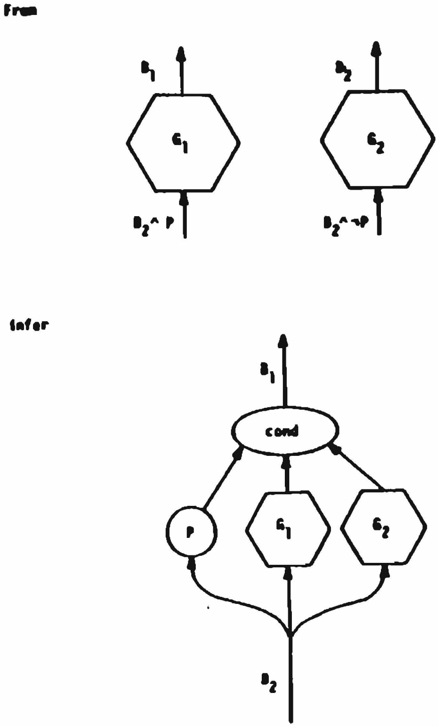


Figure 5-30: Composition rule for cond

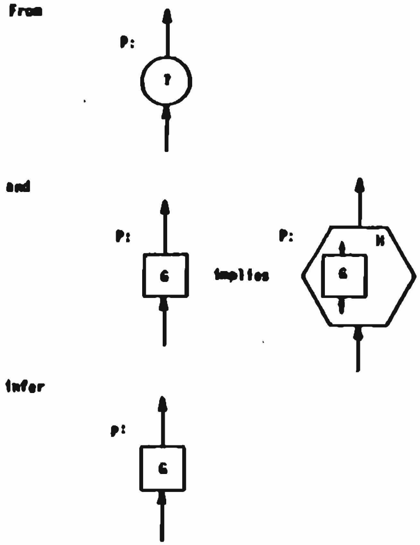


Figure 5-31: Fixed point induction rule

production with antecedent G in Figure 2-19, the rule says that to prove some property P for $H^\infty(G)$, it suffices to prove

1. Basis: P(?), where ? represents the function which always has the value undefined.
2. Induction step: For arbitrary f, assuming P(f), show P(H(f)).

If we were to view a recursive production as an application of a graph to a value, as discussed in Section 5.2, then fixed point induction becomes a case of data induction, with the program (i.e. function graph) as data.

Fixed point induction on functions sometimes fails to prove <u>defining</u> properties of functions. For example, if we were to attempt to use it on the sum_stream example above, we would take P(f) to be $f(x_1, x_2, \ldots.) = x_1,$

x_1+x_2, $x_1+x_2+x_3$,.... However, fixed point induction would fail since the basis, P(?) is false. (It is interesting to note, however, that the induction step succeeds.)

On the other hand, fixed point induction is often useful for proving properties possessed by a function other than the defining properties. We conclude this section with an example.

Example: Let AS abbreviate the function add_streams defined in Figure 2-9 and let SS abbreviate the function sum_stream defined in Figure 2-1. Suppose we wish to prove the following:

Theorem: For all solid streams x, y,

$$SS(AS(x, y)) = AS(SS(x), SS(y))$$

By a <u>solid</u> <u>stream</u>, we mean a one-level stream in which no component can be ?. For convenience, we re-define cons to be semi-strict, that is cons(?, x) is equated with ? for every x. The reason for doing so is that the transformations which follow fail without this re-definition. This does not preclude the possibility of a stream which is incomplete at the end, e.g. cons(a, cons(b, ?)) is the solid stream ab...

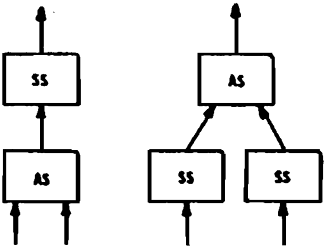


Figure 5-32: Two graphs to be shown equivalent

A graphical presentation of the theorem is given by asserting the equivalence of the two graphs shown in Figure 5-32. We shall prove the theorem using fixed point induction.

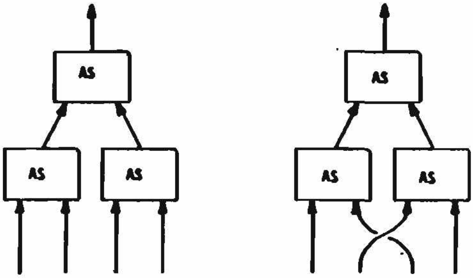


Figure 5-33: Two graphs assumed to be equivalent

During the course of the theorem, we shall appeal to the equivalence of the two graphs in Figure 5-33. The latter equivalence can be proved in a manner analogous to the theorem, but the proof is much simpler.

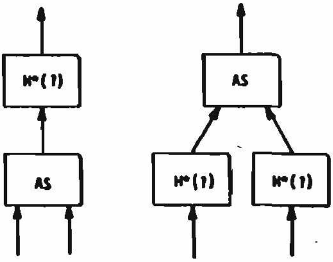


Figure 5-34: Graphs to be shown equivalent by fixed point induction, where H(g) is the consequent of g in Figure 5-3b.

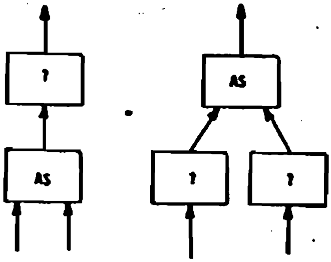


Figure 5-35: Basis of the fixed point induction

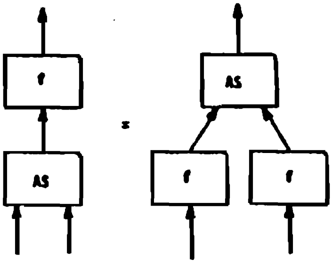


Figure 5-36: Inductive hypothesis of the fixed point induction

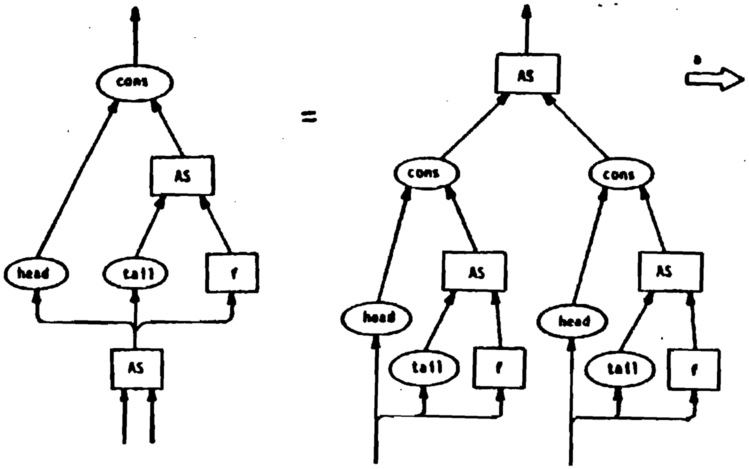


Figure 5-37: Inductive conclusion of the fixed point induction. $\xrightarrow{a}$ leads to the transformed graph in Figure 5-38.

We appeal to the fixed point induction principle to prove the equivalence of the graphs in Figure 5-34. The basis is the equivalence of the graphs in Figure 5-35, where ? is the constant function whose value is the null stream. The equivalence of these graphs follows from the definition of AS, since AS(?, ?) = cons(?, ?) = ?, according to our re-definition of cons.

The induction step assumes the equivalence of the graphs in Figure 5-36 and proves the equivalence of those same graphs, except with *f* replaced by *M*(*f*). The results of these replacements are shown in Figure 5-37.

The left graph in Figure 5-37 is shown equivalent to the right one by the series of transformations in Figure 5-38. The justifications are as follows:

a. Definition of AS.

b. Definition of head, tail.

c. Equivalence in Figure 5-33.

d. Definition of AS.

e. Inductive hypothesis.

f. Folding.

g. Definition of AS and folding.

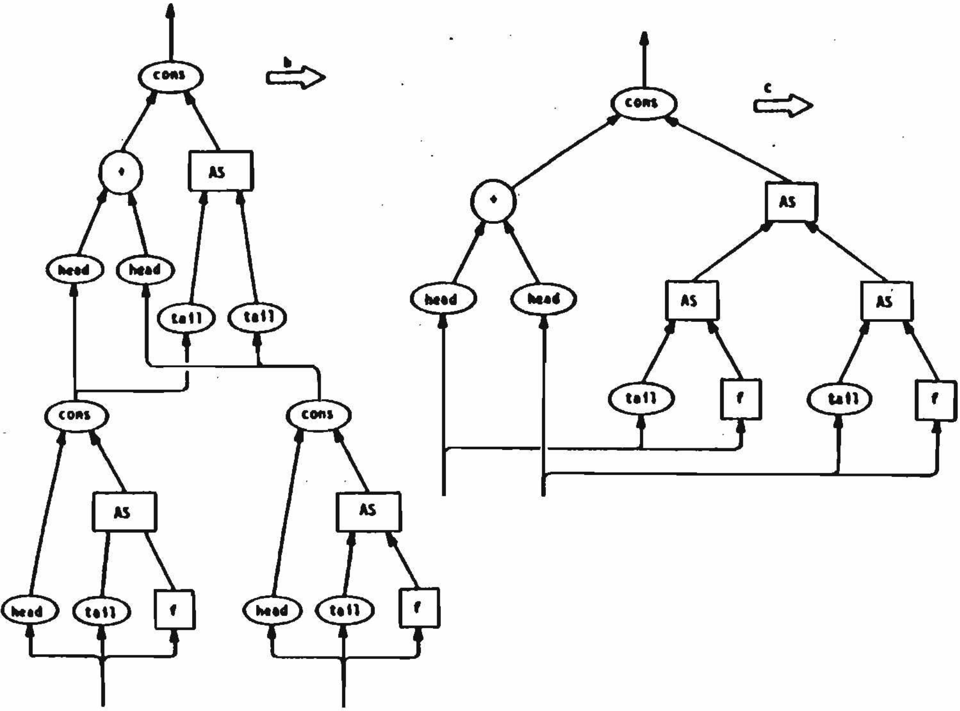
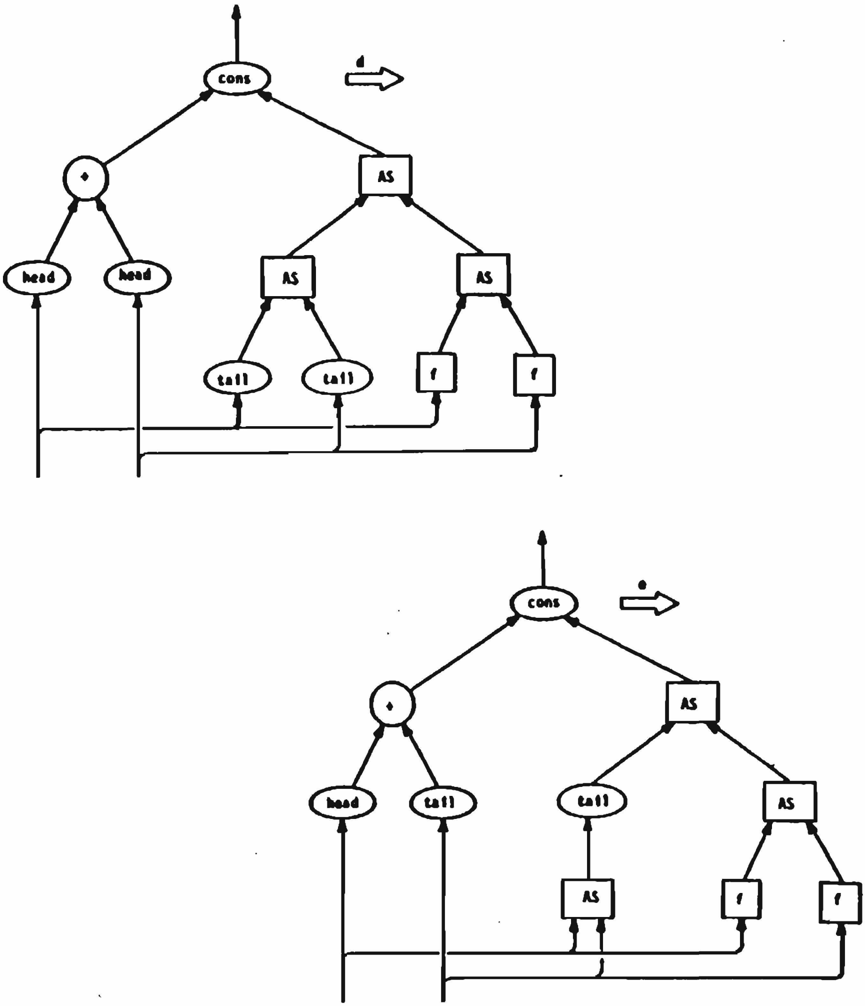
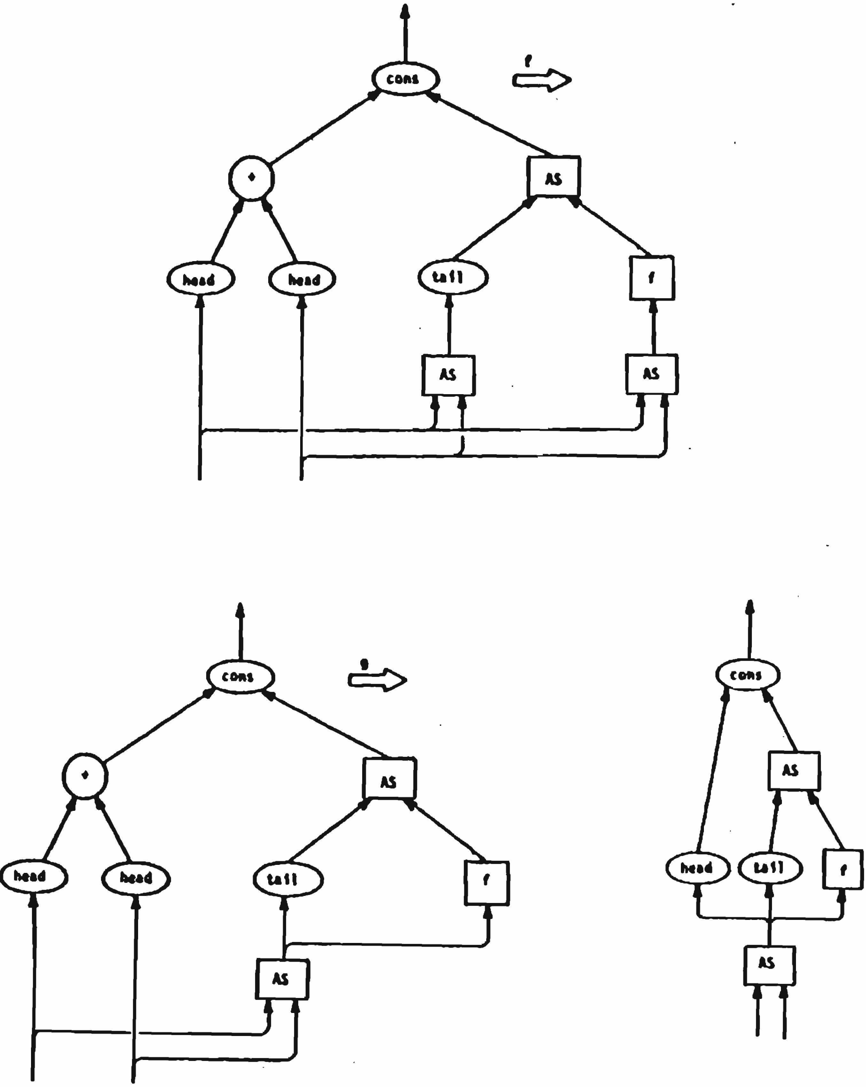


Figure 5-38: Transformations used in deriving the inductive conclusion (continued next 2 pages)


cons
d
+
AS
head
head
AS
AS
tail
tail
f
f
cons
e
+
AS
head
tail
tail
AS
AS
f
f


cons
+
head
head
tail
AS
f
AS
AS
f
cons
+
head
head
tail
AS
f
AS
g
cons
head
tail
f
AS
AS

6. Postlude

6.1. Additional Historical Material

The literature of engineering sciences, particularly electrical engineering and control theory, has seen many uses of graphical models for function-based systems. See, for example, [Zadeh and Desoer 63], which discusses a version of the determinacy theorem for general systems. Many graphical models for data-flow ([Constantine 68], [Adams 68], [Rodriguez 69], [Seror 70]) have been described in the computer science literature, the original of which seems to be [Karp and Miller 66]. In most of these, the graphs have played a rather static role, instead of being dynamically structurable entities. Many of these static models are surveyed in [Baer 78]. A different category of model is based on state-transition behavior. These models are not surveyed here, but examples may be found in [Petri 66], [Karp and Miller 69], and [Keller 76].

[Church 41] introduced the lambda-calculus, on which many models of functional programming are based. The graph model presented here is more general in that it provides a looping structure which cannot be directly represented in the lambda calculus. [Brown 62] prophesies the use of applicative languages for the exploitation of parallel processing capability. [Bohm 66] discusses the relationship between a graphical model and recursion equations. [Patil 67] discusses parallel evaluation in a graphical lambda calculus model.

The fixed point theory is due to [Kleene 52] with subsequent generalization by Scott, for example [Scott 70, 71, 76]. [Patil 70] presents a determinacy theorem for one-level stream-based systems which is similar to a related proof in [Zadeh and Desoer 63]. [Kahn 74] discusses fixed-point semantics in a model which could be considered either graphical or equational, but without the richness of Lisp operators and data structures. The latter were introduced into a graph model in [Keller 77]. [Adams 68] presents a model with Lisp-like operators, but having a semantics much less rich than the one presented here. Systems based on equations, without functions as data objects, are discussed in [O'Donnell 77]. [Turner 79] uses a related graph model to represent recursion.

The use of applicative languages to implement unbounded structures has been described in [Landin 64], [Kahn 74], [Burge 75], [Friedman and Wise 76], [Henderson and Morris 76]. The last two give sketches of correctness proofs for their evaluators, which are sequential. [Vuillemin 74] discusses issues of optimality of evaluation rules for recursive functions. [Buneman, et al. 80] describes the use of a functional language and lazy evaluation in database applications. Other aspects of applicative languages are discussed, for example, in [Landin 65], [Evans 68], [Backus 78], [Iverson 79], and [Sleep 80].

As this manuscript was being revised, [Henderson 80] made its appearance. It is a highly-recommended book, with additional examples of the use of indeterminacy and use of functions as values. Graphs are used to a limited extent, but their evaluation is executed differently than we have suggested, and the notion of enveloping is not used.

Graph models have long held appeal for representing computing systems in which the processing load is distributed among distinct physical units. The thrust of most work on distributed processing has been in the direction of process-based systems, i.e. those involving the intercommunication of multiple sequential processes [Conway 63], [Dijkstra 68], [Kahn and MacQueen 77], [Hoare 78], etc. Lately, there has been increased interest in what might be termed task-based systems. Instead of using "task" as a synonym for "process", we propose adopting a different sense of the former: a fundamental unit of work involving the computation of some atomic function. Hence task-based systems generally lend themselves to the expression of a finer grain of concurrency that do process-based systems.

Task-based systems have been discussed [Dennis 69], [Friedman and Wise 78], [Hewitt 77], and [Hewitt and Baker 78], although more work seems to have been done on high-level languages than at the implementation level. [Arvind and

Gostelow 77], [Davis 78a], and [Dennis and Misunas 74], describe some implementation aspects of these systems. The conversion of conventional programs to data flow programs for the purpose of extracting parallelism is the subject of [Urschler 73]. An implementation of FGL has been discussed in [Keller, Lindstrom, and Patil 79].

[Greif 75] and [Francez 79] discuss proof methods for task-based systems. [Park 70], [Manna 74], and [Stoy 77], among others, discuss proofs for general models representable by fixed point semantics. A proof method based on tail recursion is presented in [Mazurkiewicz 71]. [Milner 72] describes a mechanization of fixed-point induction. [Boyer and Moore 75] discuss mechanization of data induction in Lisp programs.

6.2. Conclusions

We have presented a general graph model based on functions over data types and indicated how the model can be used to represent dynamically-structured parallel and recursive computations, including intercommunication between computing modules. Proof methods and various types of transformations were discussed. We also indicated how the graphs themselves could be used as data objects.

Although this model has been found useful in developing an execution model for a highly concurrent machine architecture, we are also exploring variations of it as both a hardware and software development tool. Although other graph models have been proposed in these contexts, we feel that function based models are particularly relevant, since rather than just employing graphs as a syntactic entity, our model can also assign a semantic interpretation to each graph. This feature is extremely useful in progressive refinement, since it can avoid having to switch models as the level of description becomes more detailed.

We have avoided advocating the use of a graphical medium as the sole means of communication. A textual version of our FGL has been developed [Keller, et al. 80] and seems more useable for communicating programs once they are developed. However, the usefulness of a graphical presentation for initial development and enhancing conceptual understanding cannot be denied.

Preliminary work has been done in the use of a graphical formalism in proofs of correctness. Such a formalism offers the advantage of better visualization over conventional linear formula representations, which are prone to errors. For example, we hope to apply the technique to proofs of storage management algorithms. An initial attempt at formalizing this application appears in [Nori 79].

7. Acknowledgments

I am indebted to many people for discussions over the last several years relating to the content of this paper, including Kim Gostelow, Gilles Kahn, Gary Lindstrom, and Suhas Patil. I also appreciate the help from Jed Marti in programming the first compiler for what later became textual FGL, to Barat Jayaraman for extending and helping maintain that compiler, and to Gary Lindstrom for programming the first FGL run-time system. Additional help was provided by Francis Hunt, Anil Nori, Russ Rambo, and David Rose. Paul Hudak found a serious error in the manuscript. I have enjoyed encouragement from Arvind, Ed Ashcroft, Steve Boll, Wesley Clark, Al Davis, Jack Dennis, Paul Drongowski, Dan Friedman, Christian Gram, Tony Hearn, Dick Kieburtz, Mark Miller, Steve Muchnick, Elliott Organick, Don Oxley, Avi Silberschatz, Ronan Sleep, and Horst Wedde. A rough draft of this paper was used in a seminar at the Institute for Software Technology, GMD, Bonn, in the Spring of 1980, and I appreciate the comments and suggestions made by the participants thereof. Last, but not least, I thank the referees of the ACM Computing Survey for not letting a poorly-motivated draft get through. Of course, none of these individuals is responsible for errors or lack of clarity in this presentation.

8. References

Reference tags followed by * are not cited in the text.

[Adams 68] D.A. Adams. A computation model with data flow sequencing. Stanford University, Computer Science Dept., Tech. Rept. CS117 (1968).

[Allen 78] J. Allen. Anatomy of Lisp. McGraw-Hill (1978).

[Arvind and Gostelow 77] Arvind and K.P. Gostelow. A computer capable of exchanging processors for time. Proc. IFIP '77, 849-853 (June 1977).

[Arvind, et al. 77] Arvind, K.P. Gostelow, and W. Plouffe. Indeterminacy, monitors, and dataflow. Operating Systems Review, 11, 5, 159-169 (Nov. 1977).

[Arvind and Gostelow 78] Arvind and K.P. Gostelow. Some relationships between asynchronous interpreters of a dataflow language. In E.J. Neuhold (ed.), Formal description of programming concepts,95-119, North-Holland(1978).

[Ashcroft and Wadge 77] E.A. Ashcroft and W.W. Wadge. Lucid, a nonprocedural language with iteration. CACM, 20, 7, 519-526 (July 1977).

[Backus 78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. CACM, 21, 8, 613-641 (Aug. 1978).

[Baer 78] J.L. Baer. Graph models in programming systems. In Chandy and Yeh (eds.), Current trends in programming methodology, III, 168-231, Prentice-Hall (1978).

[Balzer 71] R.M. Balzer. Ports - A method for dynamic interprogram communication and job control. AFIPS Proc. 38, 485-489 (Spring 1971).

[Bohm 66] C. Bohm. The CUCH as a formal description language. In T.B. Steel (ed.), Formal language description languages. North-Holland (1966).

[Boyer and Moore 75] R.S. Boyer and J.S. Moore. Proving theorems about Lisp functions. JACM, 22, 1, 129-144 (Jan. 1975).

[Brown 62] G. Brown. A new concept in programming, in M. Greenberger (ed.), Management and the computer of the future. Wiley (1962).

[Buneman, et al. 80] O.P. Buneman, R.E. Frankel, and R. Nikhil. An implementation technique for database query languages. to appear in ACM TODS (1980).

[Burge 75] W.H. Burge. Recursive programming techniques. Addison-Wesley (1975).

[Chamberlin 71] D.D. Chamberlin. The single-assignment approach to parallel processing. AFIPS Proc., 263-269 (Fall 1971).

[Church 1941] A. Church. The calculi of lambda-conversion. Princeton University Press (1941).

[Constantine 68] L.L. Constantine. Control of sequence and parallelism modular programs. AFIPS Proc., 409-414 (Spring 1968).

[Conway 63] M.E. Conway. Design of a separable transition-diagram compiler. CACM, 6, 396-408 (1963).

[Cornish 79] M. Cornish. The TI data flow architectures: the power of concurrency for avionics. IEEE Third Digital Avionics System Conference (1979).

[Cornish 80] M. Cornish. Data flow control: a "motherboard" for VHSIC architecture. IEEE Workshop on Microprocessors in Military and Industrial Systems (1980).

[Curry and Feys 19] H.B. Curry and R. Feys. Combinatory Logic, I, North-Holland (1958).

[Davis 78a] A.L. Davis. The architecture and system method of DDM-1: A recursively-structured data driven machine. Proc. Fifth Annual Symposium on Computer Architecture [1978).

[Davis 78b] A.L. Davis. Data driven nets: A maximally concurrent, procedural, parallel process representation for distributed control systems. Tech. Rept. UUCS-78-108, Univ. of Utah, Dept. of Computer Science (July 1978).

[Dennis 69] J.B. Dennis. Programming generality, parallelism, and computer architecture. Proc. IFIP 68, 484-492, North-Holland (1969).

[Dennis 74] J.B. Dennis. First version of a data flow procedure language. in B. Robinet (ed.), Programming Symposium, Lecture Notes in Computer Science, 19, 362-376 (1974).

[Dennis and Misunas 74] J.B. Dennis and D.P. Misunas. A preliminary architecture for a basic data-flow processor. Proc. 2nd Annual Symposium on Computer Architecture, 126-132 (Dec. 1974).

[DeRemer and Kron 76] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. IEEE Trans., SE-2, 2 (June 1976).

[Dijkstra 68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys(ed.), Programming languages, Academic Press (1968).

[Dijkstra 76] E.W. Dijkstra. A discipline of programming. Prentice-Hall (1976).

[Evans 68] A. Evans, Jr. PAL- a language designed for teaching programming linguistics. Proc. ACM Nat. Conf., 395-403 (1968).

[Floyd 67] R.W. Floyd. Assigning meanings to programs. Proc. Symp. in Appl. Math.,19, 19-32, AMS (1967).

[Forrester 61] J.W. Forrester. Industrial dynamics. MIT Press (1961).

[Francez 79] On achieving distributed termination. in Kahn (ed.), Semantics of concurrent computation. Springer Lecture Notes in Computer Science, 70, 300-315 (1979).

[Friedman and Wise 76] D.P. Friedman and D.S. Wise. CONS should not evaluate its arguments. in Michaelson and Milner (eds.), Automata, Languages, and Programming, 257-284, Edinburgh University Press (1976).

[Friedman and Wise 77] D.P. Friedman and D.S. Wise. Aspects of applicative programming for file systems. Sigplan Notices, 12, 3, 41-55 (March 1977).

[Friedman and Wise 78] D.P. Friedman and D.S. Wise. The impact of applicative programming on multiprocessing. IEEE Trans. on Computers, C-27, 4, 289-296 (April 1978).

[Friedman and Wise 79] D.P. Friedman and D.S. Wise. An approach to fair applicative multiprogramming. in Kahn (ed.), Semantics of concurrent computation. Springer Lecture Notes in Computer Science, 70, 203-225 (1979).

[Giordano 80] J.V. Giordano. A transformation for parallel programs with indeterminate operators. UC Irvine Dataflow Architecture Project Note No. 46 (Feb. 1980).

[Gorn 59] S. Gorn. Common programming language task, Part 1, Section 5. Final Rept. AD59UR1, U.S. Army Signal Corps, Moore School of Electrical Engineering (Aug. 1959).

[Greif 75] I. Greif. Semantics of communicating parallel processes. MIT Project MAC TR-154 (Sept. 1975).

[Gurd and Watson 77] J. Gurd and I. Watson. A multilayered data flow computer architecture. Proc. 1977 International Conference on Parallel Processing (Aug. 1977).

[Guzman and Segovia 76] A. Guzman and R. Segovia. A parallel reconfigurable Lisp machine. Proc. International Conference on Information Sciences and Systems, University of Patras, Greece, 207-211 (Aug. 1976).

[Hearn 74] A.C. Hearn. Reduce 2 symbolic mode primer. Utah Symbolic Computation Group, Operating Note 5.1 (Oct. 1974).

[Hebalkar and Zilles 79] P.G. Hebalkar and S.N. Zilles. Graphical representation and analysis of information systems design. IBM

Research Rept. RJ 2465 (Jan. 1979).

[Henderson 75]* D.A. Henderson, Jr. The binding model: A semantic base for modular programming systems. MIT MAC TR-145 (Feb. 1975).

[Henderson and Morris 76] P. Henderson and J.H. Morris, Jr. A lazy evaluator. Proc. Third ACM Conference on Principles of Programming Languages, 95-103 (1976).

[Henderson 80] P. Henderson. Functional programming. Prentice-Hall (1980).

[Hennessy and Ashcroft 77] M.C.B. Hennessy and E.A. Ashcroft. Parameter-passing mechanisms and nondeterminism. Proc. Ninth ACM Symposium on Theory of Computing, 306-311 (1977).

[Hewitt 77] C. Hewitt. Viewing control structures as patterns of passing messages. Artificial Intelligence, 8, 3, 323-364 (June 1977).

[Hewitt and Baker 78] C. Hewitt and H. Baker. Actors and continuous functionals. in E.J. Neuhold (ed.), Formal description of programming concepts, 365-390, North-Holland (1978).

[Hoare 69] C.A.R. Hoare. An axiomatic basis for computer programming. CACM, 12, 10, 576-580, 583 (Oct. 1969).

[Hoare and Wirth 73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. Acta Informatica, 2, 335-355 (1973).

[Hoare 78] C.A.R. Hoare. Communicating sequential processes. CACM, 21, 8, 666-677 (Aug. 1978).

[Iverson 79] K.E. Iverson. Operators. ACM Toplas, 1, 2, 161-176 (Oct. 1979).

[Jayaraman and Keller 80] B. Jayaraman and R.M. Keller. Resource control in a demand-driven data-flow model. Proc. International Conference on Parallel Processing, 118-127 (1980).

[Johnson, et al. 80] D. Johnson, et al. Automatic partitioning of programs in multiprocessor systems. Digest of papers, IEEE Compcon 80, 175-178 (Feb. 1980).

[Johnston 69] J.B. Johnston. Structure of multiple activity algorithms. Third Annual Princeton Conference on Information Science and Systems, 38-43 (March 1969).

[Johnston 71] J.B. Johnston. The contour model of block structured processes. Sigplan Notices, 6, 2, 55-82 (Feb. 1971).

[Kahn 74] G. Kahn. The semantics of a simple language for parallel programming. Proc. IFIP '74, 471-475 (1974).

[Kahn and MacQueen 77] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. Proc. IFIP '77, 993-998 (1977).

[Karp and Miller 66] R.M. Karp and R.E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. SIAM J. Appl. Math, 14, 6, 1390-1411 (Nov. 1966).

[Karp and Miller 69] R.M. Karp and R.E. Miller. Parallel program schemata. J. Comp. and Syst. Sci., 3, 2, 147-195 (May 1969).

[Keller 73] R.M. Keller. Parallel program schemata and maximal parallelism. JACM, 20, 3, 514-537 (July 1973) and JACM, 20, 4, 696-710 (Oct. 1973).

[Keller 74] R.M. Keller. Towards a theory of universal speed-independent modules. IEEE Trans. on Computers, C-23, 1, 21-33 (Jan. 1974).

[Keller 75] R.M. Keller. Look-ahead processors. Computing Surveys, 7, 4, 177-195 (Dec. 1975).

[Keller 76] R.M. Keller. Formal verification of parallel programs. CACM, 19, 7, 371-384 (July 1976).

[Keller 77] R.M. Keller. Semantics of parallel program graphs. University of Utah, Dept. of Computer Science, Tech. Rept. UUCS-77-110 (July 1977).

[Keller 78a] R.M. Keller. Denotational models for parallel programs with indeterminate operators. In E.J. Neuhold (ed.), *Formal description of programming concepts*, 337-366, North-Holland (1978).

[Keller 78b] R.M. Keller. An approach to determinacy proofs. University of Utah, Dept. of Computer Science, Tech. Rept. UUCS-78-102 (March 1978).

[Keller, Lindstrom, and Patil 79] R.M. Keller, G. Lindstrom, and S. Patil. A loosely-coupled applicative multi-processing system. AFIPS Proc. (June 1979).

[Keller, Lindstrom, and Patil 80] R.M. Keller, G. Lindstrom, and S. Patil. Data-flow concepts for hardware design. Digest of papers, IEEE Compcon 80, 105-111 (Feb. 1980).

[Keller 80] Divide and CONCer: Data structuring for applicative multiprocessing. Proc. 1980 Lisp Conference, 196-202 (Aug. 1980).

[Keller, et al. 80] FGL Programmer's guide. Unpublished memo.

[Keller and Lindstrom 80] R.M. Keller and G. Lindstrom. Hierarchical analysis of a distributed evaluator. Proc. International Conference on Parallel Processing, 299-310 (1980).

[Kleene 52] S.C. Kleene. *Introduction to metamathematics*. Van Nostrand (1952).

[Kleene 56] S.C. Kleene. Representation of events in nerve nets and finite automata. in C.E. Shannon and J. McCarthy (eds.), Automata studies, 3-42, Princeton University Press (1956).

[Kosinski 79] P.R. Kosinski. Denotational semantics of determinate and non-determinate data flow programs. MIT/LCS/TR-220 (May 1979).

[Landin 64] P.J. Landin. The mechanical evaluation of expressions. Computer J., 6, 4, 308-320 (Jan. 1964).

[Landin 65] P.J. Landin. A correspondence between Algol 60 and Church's Lambda-Notation, CACM, 8, 2, 89-101 (Feb. 1965) and 8, 3, 158-165 (Mar. 1965).

[Manna 74] Z. Manna. Mathematical theory of computation. McGraw-Hill (1974).

[Mazurkiewicz 71] A.W. Mazurkiewicz. Proving algorithms by tail functions. Information and Control 18, 220-226 (Apr. 1971).

[McCarthy 60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, I. CACM, 5, 27-37 (1970).

[McCarthy 63a] J. McCarthy. Towards a mathematical science of computation. IFIP '62, Proc. 21-28 (1963).

[McCarthy 63b] J. McCarthy. A basis for a mathematical theory of computation. in P. Braffort and D. Hirshberg (eds.), Computer programming and formal systems, 33-70, North-Holland (1963).

[Milner 72] R. Milner. Implementation and applications of Scott's logic for computable functions. Sigplan Notices, 7, 1, 1-6 (Jan. 1972).

[Morris, et al. 80]* J.H. Morris, E. Schmidt, and P. Wadler. Experience with an applicative string processing language.

[Morris and Schwarz 80]* L. Morris and J. Schwarz. Computing cyclic list structures. Proc. 1980 Lisp Conference, 144-153 (Aug. 1980).

[Moses 70] J. Moses. The function of FUNCTION in Lisp. SIGSAM Bulletin, 13-27 (July 1970).

[Mudge 78] T.N. Mudge. A data driven computer architecture. Proc. Johns Hopkins Conf. on Information Sciences and Systems (1978).

[Nori 79] A.K. Nori. A storage reclamation scheme for AMPS. M.S. Thesis, Dept. of Computer Science, University of Utah (Dec. 1979).

[O'Donnell 77] M. O'Donnell. Computing in systems described by equations. Lecture Notes in Computer Science, 58 (1977).

[Park 70] D. Park. Fixpoint induction and proofs of program correctness. Machine Intelligence, 5, 279-299, North-Holland (1970).

[Peterson and Hewitt 70] M. Paterson and C. Hewitt. Comparative schematology. Rec. Project MAC Conference on Concurrent Systems and Parallel Computation, 119-128 (1970).

[Patil 67] S. Patil. An abstract parallel-processing system. M.S. Thesis, MIT Dept. of Electrical Engineering (June 1967).

[Patil 70] S. Patil. Closure Properties of interconnections of determinate systems. Proc. Project MAC Conference on Concurrent Systems and Parallel Computation, 107-116 (June 1970).

[Patri 66] C.A. Patri. Communication with automata. Supplement 1 to Tech. Rept. RADC-TR-65-377. Griffiss Air Force Base, New York (1966).

[Plas 76] A. Plas, et al. LAU system architecture: A parallel data-driven processor based on single assignment. Proc. International Conference on Parallel Processing, 293-302 (Aug. 1972).

[Plotkin 76] G.D. Plotkin. A power domain construction. SIAM J. Comp. 5, 3, 452-487 (Sept. 1976).

[Pritsker and Pegden 79] A.A.B. Pritsker and C.D. Pegden. Introduction to simulation and Slam. Halsted Press (1979).

[Pugh 70] A.L. Pugh III. Dynamo user's manual. MIT Press (1970).

[Quine 60] W. Quine. Word and object. Wiley (1960).

[Rabiner and Rader 72] L.R. Rabiner and C.M. Rader. Digital signal processing. IEEE Press (1972).

[Reeker 71] L.H. Reeker. State graphs and context free languages. in Kohavi and Paz (eds.), Theory of machines and computations, 143-151, Academic Press (1971).

[Ritchie and Thompson 75] D.M. Ritchie and K. Thompson. The Unix time-sharing system. CACM, 17, 7, 365-381 (July 1975).

[Rodriguez 69] J.E. Rodriguez. A graph model for parallel computation. MIT Project MAC Tech. Rept. TR-64 (1969).

[Rosen 73] B. Rosen. Tree manipulating systems and Church-Rosser theorems. JACM, 20, 1, 160-187 (Jan. 1973).

[Rosen 75]* B.K. Rosen. Deriving graphs from graphs by applying a production. Acta Informatica 4, 337-357 (1975).

[Ross 77] D.T. Ross. Structured analysis (SA): A language for communicating ideas. IEEE Trans., SE-6, 1, 16-33 (Jan. 1977).

[Rumbaugh 77] J. Rumbaugh. A data flow multiprocessor. IEEE Transactions, C-26, 2, 138-146 (Feb. 1977).

[Schwarz 77]* J. Schwarz. Using annotations to make recursion equations behave. Res. Rept. 43, Dept. of Artificial Intelligence, University of Edinburgh (Sept. 1977).

[Scott 70] D. Scott. Outline of a mathematical theory of computation. Fourth Annual Princeton Conference on Information and Systems Sciences, 169-176 (March 1970).

[Scott 71] D. Scott. The lattice of flow diagrams. Symposium on semantics of algorithmic languages, 188, 311-366, Springer-Verlag (1971).

[Scott 76] D. Scott. Data types as lattices. SIAM J. Comp., 5, 3, 522-587 (Sept. 1976).

[Seror 70] D. Seror. DCPL: A distributed control programming language. Tech. Rept. UTEC-CSc-70-108, Dept. of Computer Science, University of Utah (Dec. 1970).

[Sleep 80] M.R. Sleep. Applicative languages, dataflow, and pure combinatory code. Digest of papers, IEEE Compcon 80, 112-115 (Feb. 1980).

[Smith and Chang 75] J.M. Smith and P.Y.T. Chang. Optimizing the performance of a relational algebra database interface. CACM 18, 10, 568-579 (Oct. 1975).

[Smoliar 79] S.W. Smoliar. Using applicative techniques to design distributed systems. Proc. IEEE Conf. on Reliable Software (April 1979).

[Smyth 78] M.B. Smyth. Power domains. JCSS, 16, 23-36 (1978).

[Stoy 77] J. Stoy. The Scott-Strachey approach to the mathematical semantics of programming languages. MIT Press (1977).

[Syre, et al. 77] J.C. Syre, et al. Pipelining, parallelism and asynchronism in the LAU system. Proc. International Conf. on Parallel Processing, 87-92 (Aug. 1977).

[Tesler and Enea 68] L.G. Tesler and H.J. Enea. A language design for concurrent processes. AFIPS Proc., 403-408 (Spring 1968).

[Treleaven and Mole 80]* P.C. Treleaven and G.F. Mole. A multi-processor reduction machine for user-defined reduction languages. Proc. 7th Annual Symp. on Computer Architecture (May 1980).

[Turner 79] D.A. Turner. A new implementation technique for applicative languages. Software - Practice and Experience, 9, 31-49 (1979).

[Urschler 73] G. Urschler. The transformation of flow diagrams into maximally parallel form. Proc. Sagamore Conference on Parallel Processing, 38-46 (March 1973).

[Vuillemin 74] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. JCSS, 9, 332-354 (1974).

[Wadsworth 76] C.P. Wadsworth. The relation between computational and denotational properties for Scott's models of the lambda-calculus. SIAM J. Comput., 5, 3 (Sept. 1976).

[Ward 74] S.A. Ward. Functional domains of applicative languages. MIT Project MAC, Rept. MAC TR-136 (Sept. 1974).

[Ward and Halstead 80]* S.A. Ward and R.H. Halstead, Jr. A syntactic theory message passing. JACM 27, 2, 365-383 (Apr. 1980).

[Weinberg 78] V. Weinberg. Structured analysis. Prentice-Hall (1978).

[Weng 79] K.S. Weng. An abstract implementation for a generalized data flow language. MIT LCS Rept. TR-228 (May 1979).

[Wolfberg 72]* M.S. Wolfberg. Fundamentals of the Ambit/L list-processing language. Sigplan Notices, 7, 10, 66-75 (Oct. 1972).

[Yourdon and Constantine 79] E. Yourdon and L.L. Constantine. Structured design. Prentice-Hall (1979).

[Zadeh and Desoer 63] L.A. Zadeh and C.A. Desoer. Linear system theory. McGraw-Hill (1963).