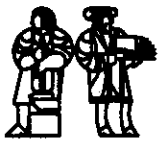
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Fixpoint Equations and Dataflow

Computation Structures Group Memo 256
December 1985

Keshav K. Pingali

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

List of Figures

1. Fixpoint Equations

Definition 1: Let f:D → D be a function that maps a set D to itself. An equation of the form $x = f(x)$ is called a *fixpoint equation.* If x_f is an element of D that satisfies this equation, it is said to be a *fixpoint* of f.

□

A function f:D → D can have zero, one or many fixpoints.

Example Consider f: $\mathbb{R} \rightarrow \mathbb{R}$ where f(x) is $4x^2 + 2$. Verify that f has no fixpoints *i.e.*, the equation $x = 4x^2 + 2$ has no solution in $\mathbb{R}$.

How many fixpoints do the trigonometric functions *sin(x)* and *tan(x)* have ?

Fixpoint equations arise naturally in computer science because a convenient way to describe an infinite set or a function is by recursive programs.

Example: Consider the following recursive program which computes McCarthy's 91 function :

```
def F(x) <= if x > 100 then (x - 10) else F(F(x + 11))
```

While this program does not look like a fixpoint equation, we can translate it into the following equation -

```
f = [λg.λx. if x > 100 then x -10 else g(g(x + 11))] f
```

The expression within square brackets has the functionality $(\mathcal{N} \rightarrow \mathcal{N}) \rightarrow \mathcal{N} \rightarrow \mathcal{N}$ - *i.e.*, it takes a function from $\mathcal{N}$ to $\mathcal{N}$ and returns a function from $\mathcal{N}$ to $\mathcal{N}$. Such functions are sometimes called *functionals* or more generally *higher-order functions.* The 91 function is a fix-point of the functional shown above.

Exercise: Verify that the following function h: $\mathcal{N} \rightarrow \mathcal{N}$ satisfies the fixpoint equation for the 91 function.

```
h  = λn. if n > 100 then (n - 10) else 91
```

To see how fixpoint equations can be used to define sets, see [4] where it is shown that the language described by a context-free grammar can be expressed as the solution to a fixpoint equation. In formal language theory, this result is known as the Ginzburg-Rice theorem.

2. Two Fixpoint Theorems

There are a wide variety of fixpoint theorems in the literature. The basic idea behind all of them is to give conditions on both the set D and the function f:D → D that are sufficient to guarantee that f has a fixpoint and that the fixpoint can be determined by some simple algorithm. Of particular interest in computer science is the case when the set D is a partially ordered set. The main result in this area is attributed to Tarski who proved a particular fixpoint theorem in the case that D is a complete lattice [7]. Of more interest to us is the case of general partial orders. Before giving the theorem, we motivate some of the technical machinery we need by proving a simpler fixpoint theorem for the case when D is a finite, partially ordered set.

Theorem 2: First Fixpoint Theorem : Let D be a finite, partially ordered set with one

least element[1] and let f:D → D be a monotonic function. For some finite $k \geq 0$, $f^k(\perp)$ is the least fixpoint of f.

Proof: Since $\perp$ is the least element of D, $\perp \subseteq f(\perp)$. From the monotonicity of f, we can conclude that $f(\perp) \subseteq f^2(\perp)$. Inductively, then, we have the following chain :

$$\perp \subseteq f(\perp) \subseteq f^2(\perp) \subseteq .... \subseteq f^i(\perp) \subseteq ....$$

However, since D is finite, there must be some $k \geq 0$ such that $f^k(\perp)$ is the largest element of the chain - *i.e.*,

$$f^k(\perp) = f^{k+1}(\perp) ....$$

Hence, $f^k(\perp)$ must be a fixpoint of f.

Moreover, if *p* is any other fixpoint of f, then $\perp \subseteq p$. Since f is monotonic, $f(\perp) \subseteq f(p) = p$. Inductively, then, for all $i \geq 0$, $f^i(\perp) \subseteq p$; in particular, $f^k(\perp) \subseteq p$. Hence, $f^k(\perp)$ is the least fixpoint of f.

□

Unfortunately, this theorem is not very useful because of the restriction that D be a *finite* set. Notice that the proof breaks down if D is an infinite set because then the chain

$$\perp \subseteq f(\perp) \subseteq f^2(\perp) \subseteq .... \subseteq f^i(\perp) \subseteq ....$$

need not have a largest element.

To prove a similar (and more useful) fixpoint theorem when D is an infinite set, we proceed as follows. A natural extension of the idea of the largest element of a chain is the idea of the *least upper bound* of a chain.

Definition 3: Let D be a partially ordered set and let $x_0 \subseteq x_1 ... \subseteq x_i \subseteq ....$ be a chain in D. $y \in D$ is said to be the *least upper bound* or lub of the chain iff :

1. for all i, $x_i \subseteq y$ - *i.e.*, y is an upper bound of the chain

2. if z is an upper bound of the chain, then $y \subseteq z$ - *i.e.*, y is the *least* upper bound.

□

Notice that this is essentially a restatement of the familiar idea of the limit of a sequence of real numbers, although in a more general context.

Example Let D be the [0, 1) interval of the real line. This is a totally ordered set. The lub of the chain $0 \subseteq 1/4 \subseteq 3/8 \subseteq 7/16$ .... is 1/2. The chain $0 \subseteq 1/2 \subseteq 3/4 \subseteq 7/8$ .... in D does *not* have a lub (notice that 1 is not a member of D). Notice that if D' is [0, 1], the chain $0 \subseteq 1/2 \subseteq 3/4 \subseteq 7/8$ .... in D' has a lub.

This example suggests that if we want to talk about lub's of infinite chains, it is convenient to

---

[1] we denote the least element by $\perp$

restrict our attention to partially ordered sets in which *every* infinite chain has a lub.

Definition 4: An ω-complete partial order is a partial order in which every chain has a lub.

□

Let us add the condition that D must be an ω-complete, partially ordered set and reconsider the chain

$$\perp \subseteq f(\perp) \subseteq f^2(\perp) \subseteq .... \subseteq f^i(\perp) \subseteq ....$$

We can now assert that this chain must have an lub, and we can denote the lub by $\cup_i f^i(\perp)$. However, we still cannot assert that it is a fixpoint of f.

Exercise Show that $f(\cup_i f^i(\perp))$ must be an upper bound of the chain $\perp \subseteq f(\perp) \subseteq f^2(\perp)$ ... We would like a stronger condition - that $f(\cup_i f^i(\perp))$ is $\cup_i f^i(\perp)$ *i.e.*, the *least* upper bound of the chain.

To ensure that $\cup_i f^i(\perp)$ is the least fix point of f, it is sufficient to require that f must not only be monotonic but also *continuous.*

Definition 5: Let D be an ω-complete partial order with one least element. $f:D \rightarrow D$ is said to be continuous iff for every chain $x_0 \subseteq x_1 \subseteq x_2$ .... in D, $f(\cup_i x_i) = \cup_i f(x_i)$.

As before, this definition of continuity corresponds closely to the definition of continuity in analysis. The following diagram may help the reader to appreciate this fact.

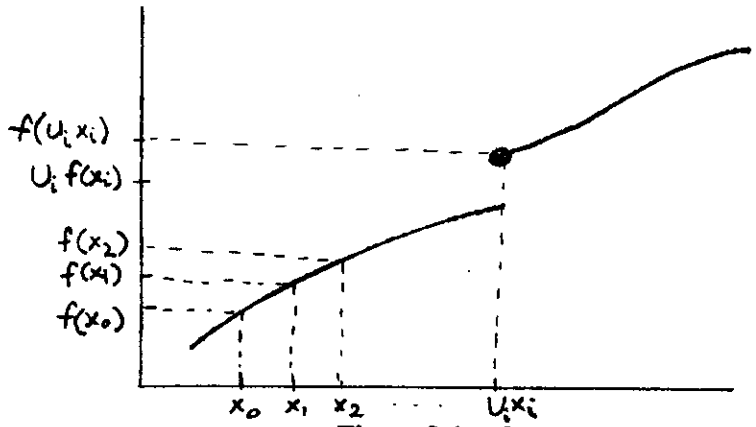


Figure 2-1: Continuity

Exercise Verify that if f is continuous, then $f(\cup_i f^i(\perp)) = \cup_i f^i(\perp)$.

Finally, we have

Theorem 6: Second Fixpoint Theorem : If D is an ω-complete, partially ordered set with one least element and $f:D \rightarrow D$ is a monotonic and continuous function, then $\cup_i f^i(\perp)$ is the least fixpoint of f.

Proof: Left as an exercise. Follows simply from the discussion above.

This method of finding the least fixpoint of f is sometimes called Kleene's iterative method.

This theorem can be extended quite easily to deal with the case when the function f can take more than one argument; for example, if f has functionality $D \times D \rightarrow D$. A further extension is to

systems of fixpoint equations of the form

```
x = f(x, y)
y = g(x, y)
```

where x and y are both elements of D. Once again, the extension is simple and the interested reader is referred to Manna [6].

3. An Application of the Second Fixpoint Theorem

The second fixpoint theorem is often used to give an abstract ("denotational") semantics for programs. For example, see Manna [6] and de Bakker [2] for expositions of how the theorem can be used to give a semantics for recursive programs such as the one for McCarthy's 91 function which was given in the introduction. We take up a different example that is more in keeping with the spirit of this course.

3.1. Networks of Communicating Processes

Consider a network of computing stations which can communicate with each other by means of message buffers. There is no global clock and there are no constraints on the relative speeds of the stations. The message buffers are idealized in the sense that they are infinite in size, no message is ever lost and messages sent by a station (say P) to a station (say Q) are received by Q in the order in which they were sent by P. Certain of the message buffers are considered to be input or output buffers on which messages can be sent into and sent out from the network. Figure 3-1 shows an example of such a network.

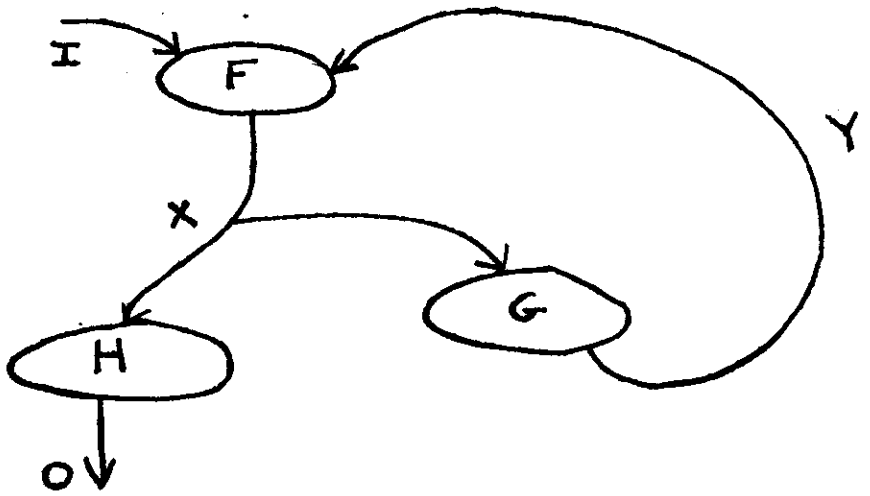


Figure 3-1: A Network of Stations Communicating via Message Buffers

An operational semantics for such a network can be given by first defining the concept of an "instantaneous state" for the entire system. Usually, this is taken to be a state vector in which there is one element for the state of each computing station and message buffer. The behavior of the network is then characterized by giving a transition (or "next-state") relation which specifies the

permissible state transitions of the system. Figure 3-2 shows an example of a state transition diagram. Notice that since stations operate asynchronously, this relation is not, in general, a function; *i.e.*, in operational terms, a system in a given state makes a transition to some state which is one of *many* possible states. Thus, the network can exhibit non-deterministic behavior. Nevertheless, it may still be possible that the network exhibits deterministic I/O behavior, in the sense that for a given set of input messages, the output messages produced by the network are uniquely defined. A natural question then is whether one can place sufficient conditions on such a network for ensuring deterministic I/O behavior.

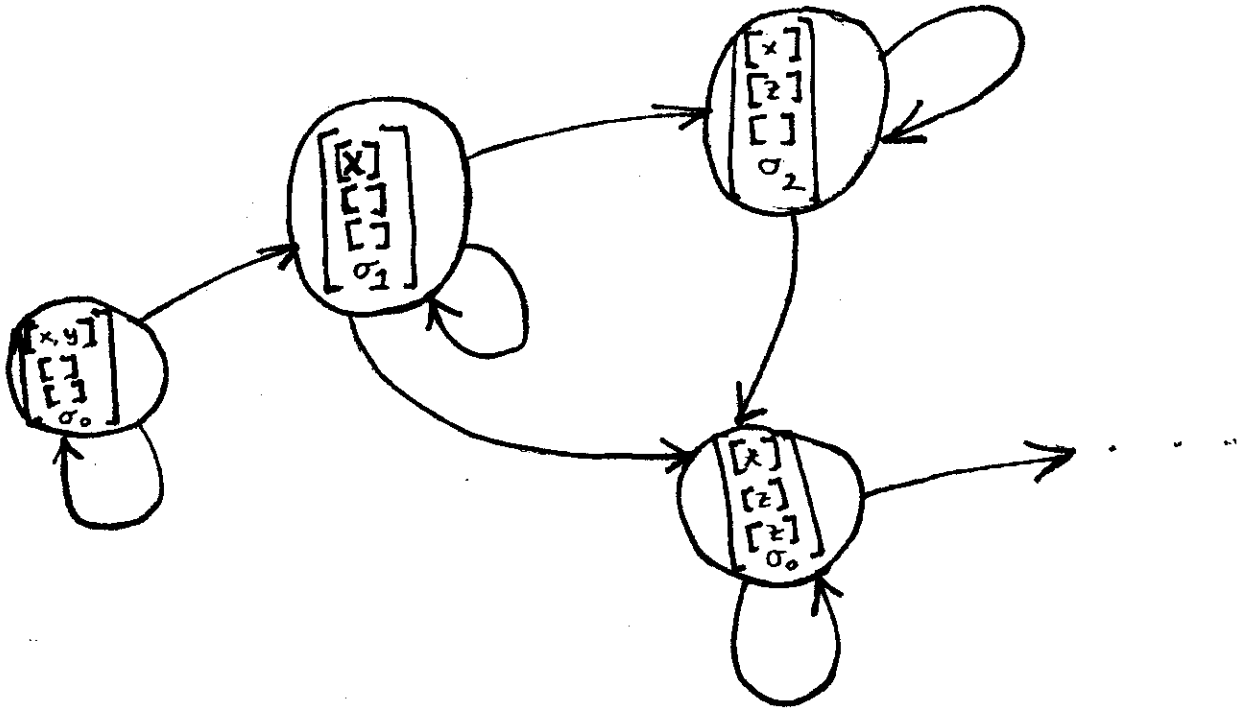


Figure 3-2: An Example of a State Transition Diagram

In 1974, Gilles Kahn gave a fairly general solution to this problem [5]. He imposed three conditions on the computing stations, two of which are the following :

1. at any given time, a station is either computing or waiting for a message on *one* of its input lines

2. each computing station follows a sequential program.

Hence, no two stations may send messages on the same buffer, and no station may "poll" two or more input buffers for messages. If a station attempts to read from an empty message buffer, it gets "stuck" until some station sends a message on that buffer. Notice that since buffers are infinite in size, a station can never get stuck while outputting a message.

Kahn then defined the *history* of a message buffer to be the sequence of messages seen by an observer monitoring the messages sent on the buffer. If every message must come from some set M

(for example the set of integers and booleans), then the set of message sequences is the set M^{ω} which is the set of all finite and denumerably infinite sequences of elements over M. The empty message sequence (written as [ ]) is assumed to be an element of this set. A natural ordering on the set of message sequences is the so-called "prefix" ordering - for example, the message sequence [3,5] (the message sequence in which the first message is 3 and the second message is 5) is below the message sequence [3,5,8] in the partial order. Note that the sequence [3,6] is incomparable to the sequence [3,5].

The third requirement that Kahn imposed on stations was :

3. under the prefix ordering on histories, each station is a monotonic and continuous function from histories to histories

Thus, with every station, we can associate a monotonic and continuous function from sequences to sequences that determines the I/O behavior of the station. The intuitive meaning of monotonicity in this context is that a station never retracts a message - giving a station more input can only cause it to produce more output (*i.e.*, send out zero or more messages). The intuitive notion of continuity is that the behavior of a station when it is given a infinite sequence of messages can be determined by the limit of its behavior on all finite prefixes of that input sequence - for example, a station cannot decide to send some output only after it receives an infinite amount of input.

Kahn asserted that a network in which every computing station satisfied these three conditions has determinate I/O behavior regardless of any "internal" non-determinacy it may have. Moreover, the I/O behavior can be determined by solving a set of fixpoint equations that we can associate with a network in a straight-forward way. For example, the set of equations for the network shown in Figure 3-1is shown below -

```
x = f(1, y)
y = g(x)
o = h(x)
```

The functions f, g and h are the (monotonic and continuous) functions from histories to histories that characterize the I/O behavior of stations F, G and H respectively. The variables x, y, i, and o represent the histories of the corresponding message buffers.

Using the fixpoint theorem, we can assert that this set of equations has a unique least solution. Kahn asserted that this solution corresponds exactly to the I/O behavior that is determined by the operational semantics[2]

Exercise : The iterative method of computing fixpoints corresponds to following a particular path in the state transition diagram of the network. Does this path correspond to some particular way of scheduling the execution of the stations? Is it a fair schedule? Prove that the output produced by any other way of scheduling stations will produce a prefix of the output specified by the least fixpoint of the semantic equations (this part is difficult - see [1, 3]).

---

[2] A historical note of some interest is that Kahn never actually proved the equivalence of the two semantics - he merely asserted that they were. A rather detailed proof of the equivalence in a slightly more general context was first given by Faustini [3].

The reader who is interested in a rigorous treatment of this discussion is referred to Kahn [5].

What makes all this interesting is that Kahn's work solved not only the problem of giving sufficient conditions to ensure determinacy, but also provided a way of determining the I/O behavior of such a network. Moreover, by relating all this to work by Scott and others, he gave some very powerful proof rules that can be used to reason about such networks, such as determining the equivalence of two networks. Of course, we should not expect any miracles - the halting problem still remains undecidable, but we at least have a framework for proving properties of programs in which irrelevant internal details of the system are hidden from the theory.

The most powerful proof rule is Scott's rule of induction :

Scott's Rule of Induction: Let D be an ω-complete partially ordered set with one least element and let $f:D \rightarrow D$ be monotonic and continuous. If $P(x:D)$ is a predicate; then if

1. $P(\perp)$ holds, and

2. $P(x) \Rightarrow P(f(x))$

then $P(\bigcup_i f^i(\perp))$.

Unfortunately, this rule is not valid for any predicate, but only for a class of predicates called admissible predicates. For a definition of these predicates, the reader is referred to Manna [6]. Notice that ordinary induction suffices to prove that $P(f^k(\perp))$ for all finite k. To make the jump to least upper bound of the sequence, we need Scott's rule.

For the kind of equations for Kahnian networks that we have been considering, we can restate Scott's rule of induction in the following, more convenient form. To avoid introducing more notation, we consider the network of Figure 3-1. Let $\mathfrak{I}$, $\mathfrak{X}$, $\mathfrak{Y}$ and $\mathfrak{O}$ be the histories of the buffers I, X, Y and O that are determined by the least fixpoint solution. Let x, y and o be the histories of lines X, Y and O at some point in the computation. Scott's rule of induction can be rephrased for this network as follows: If P(I, X, Y, O) is an admissible predicate, and

1. $P(\mathfrak{I}, [\,], [\,], [\,])$ holds and

2. $P(\mathfrak{I}, x, y, o) \Rightarrow P(\mathfrak{I}, f(\mathfrak{I}, y), g(x), h(x))$

then $P(\mathfrak{I}, \mathfrak{X}, \mathfrak{Y}, \mathfrak{O})$ holds.

In other words, to prove a property P of some network, we must show that P is true initially when the inputs are supplied and all other message buffers are empty and that if P is true at some point in the computation, then P is true at the next step after all stations have fired simultaneously. As usual, this applies only when P is admissible. In operational terms, this can be interpreted as follows. If P is true at the beginning of the computation, and P is true at every step of the "fair schedule" that corresponds to iteratively finding the fixpoint, then P is true at the end of the computation.

Exercise: What if some other schedule is followed ? Argue that P need not hold at "intermediate" steps of the computation, but that P will hold at the end of the computation as long as the schedule is a fair one.

We now demonstrate the power of this rule by showing how it can be used to prove the basic result about the Queued and Unraveling Interpreters for dataflow graphs.

3.2. An Application of the Theory in Dataflow

Our interest in Kahn's result arises from the fact that any dataflow graph can be translated trivially into Kahn's framework. For definiteness, let us consider Dennis's graphs in which every node represents an operator from the set {T-box, true-gate, false-gate, (deterministic) merge}. By making each node in the dataflow graph into a computing station and making every arc in the graph into a message buffer, we get a network of computing stations. We leave it to the reader to verify the following facts :

- for the Queued interpreter, the history of a line is a sequence of zero or more messages. If these sequences are ordered by the prefix ordering, then each operator in the graph represents a monotonic and continuous function from histories to histories.

- for the Unraveling interpreter, the situation is more complicated because the physical order in which messages are sent down a line need not correspond to the logical order of the messages. For example, the message with logical number *i* can be produced before the message at logical position *i-1*. In operational terms, we would have to tag every message with its logical position number. Once messages are tagged, the exact physical order in which messages are sent down a line becomes irrelevant, and the the history of a line can be considered to be a set of tagged messages. For example, one possible history is {<3, 1>, <4, 2>, <8, 3>} which represents a history of 3 messages in which the first message is 3, the second is 4 and the third is 8. Another possible history is {<3 , 1>, <7, 4>, <4, 2>} - in this history, the message at logical position #4 has been produced before the message at logical position #3. At some later point in the computation, the station may send out the message at the third position (say 8), and the history then is {<3, 1>, <7, 4>, <4, 2>, <8, 3>}. Intuitively, the appropriate ordering appears to be the subset ordering on sets of tagged messages. We leave it to the reader to verify that under this ordering, the operators in the language are monotonic and continuous functions from tagged message sets to tagged message sets.

A natural question one can ask is whether for every program, the Unraveling interpreter produces "more output" than the Queued interpreter, where producing "more output" corresponds to the intuitive idea of producing all the messages (with appropriate tags) produced by the Queued interpreter, and perhaps, some more tagged messages. For example, if the Unraveling interpreter produces {<5, 1>, <6, 4>, <3, 2>} and the Queued interpreter produces [5, 3], then the Unraveling interpreter has produced more output. On the other hand, if the Queued interpreter has produced [5, 3, 7], then the outputs are incomparable.

It is convenient to define a function tagged which takes a message sequence and returns a set of messages tagged appropriately. For example, tagged([3,4,6]) is {<3, 1>, <4, 2>, <6, 3>}.

It is clear that for some programs, the Unraveling interpreter produces more output than the Queued interpreter. An example of this is shown in Figure 3-3.

What must be proved is that this is the case for *all* programs. Clearly, one cannot prove this by

Figure 3-3: Difference between the Unraveling Interpreter and the Queued interpreter

giving examples - we must give a proof. The obvious thing to try is Scott's rule. Unfortunately, we have two different domains - one of message sequences for the Queued interpreter (say D1), and another of tagged message sets for the Unraveling interpreter (say D2). To use Scott's rule, we must construct one domain. This is a standard construction - the (separated) sum of two domains D1 and D2 (written as D1 + D2) is the disjoint union of all the elements of D1 and D2, together with a new $\perp$ element. The elements of the sum domain are ordered as follows -

1. $\perp \subseteq x$ for all $x \in D1 + D2$.

2. if x is an element of Di and y is an element of Dj, then $x \subseteq y$ iff $i = j$ and $x \subseteq y$ in Di.

Pictorially, the sum of the two domains can be shown as in Figure 3-4

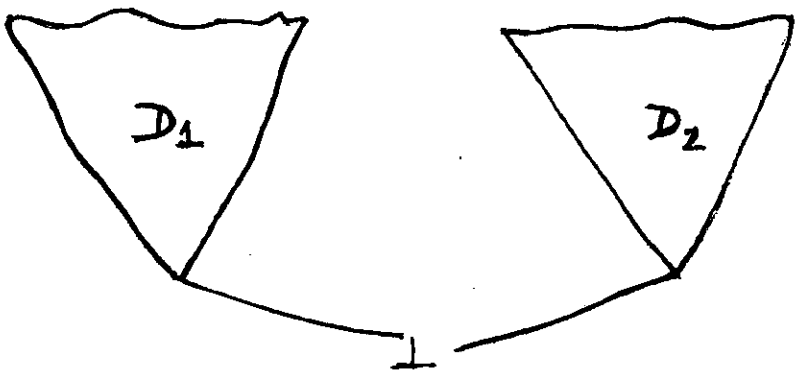


Figure 3-4: Sum of Two Domains

We can now extend any monotonic and continuous function that operates on one of the domains (say D1) to a monotonic and continuous function that operates on the sum domain D1 + D2. This is done in a fairly standard way as follows. Intuitively, f: D1 → D1 is extended to g: (D1 + D2) → (D1 + D2) where g maps elements of D1 exactly as f did, and maps $\perp$ and elements of D2 to some value that is consistent with monotonicity - for example, $\perp$. This extension is merely a mathematical construction - no deep operational meaning should be sought for.

We now define a predicate more-defined-than(x, y) where x and y are elements of (D1 + D2) -

- if x is an element of the D2 component of the domain, and y is an element of the D1

component of the domain, and tagged(y) is a subset of x, then more-defined-than(x, y) is true

- in all other cases, the predicate is false.

It can be shown that this is an admissible predicate.

To simplify matters again, we consider the network of Figure 3-1. Imagine two copies of this network placed side-by-side. We will use the Queued interpreter on one of the copies and the Unraveling interpreter on the other. Both networks are given the same inputs (tagged for the Unraveling interpreter) and allowed to compute.

We translate this operational scenario into the semantic scenario as follows. The behavior of the two networks can be described by the obvious fixpoint equations - remember that the history variables in these equations represent values in the sum domain D1 + D2.

$x_1 = f_1(i_1, y_1)$ $\quad x_2 = f_2(i_2, y_2)$
$y_1 = g_1(x_1)$ $\quad y_2 = g_2(x_2)$
$o_1 = h_1(x_1)$ $\quad o_2 = h_2(x_2)$

Consider the predicate P defined as

more-defined-than(i_2, i_1) $\wedge$ more-defined-than(x_2, x_1) $\wedge$ more-defined-than(y_2, y_1) $\wedge$ more-defined-than(o_2, o_1)

This is an admissible predicate. When both networks are given the same message inputs (tagged appropriately for the Unraveling interpreter), we can use Scott's rule as follows -

- P is trivially true when the histories i_1 and i_2 are the inputs to the networks and all other histories are [ ].

- if P is true for some histories, we verify that P will be true after all stations fire simultaneously. This is done proving that each of the terms in the disjunction is true, which is proved simply by considering the functionality of the operators. operators.

This proves the required result.

As before, the aim of this discussion was to provide the intuition behind the technical results - the reader who wants the formal details is banished to the references. We note that the original proof of this result [1] also uses Scott's rule, but uses a different set-up to sort out the various technicalities.

References

1. Arvind, and Gostelow, K. P. Some Relationships Between Asynchronous Interpreters of a Dataflow Language. In E. J. Neuhold, Ed., *Formal Description of Programming Languages*, North-Holland, New York, 1977.

2. De Bakker, J. *Foundations of Computer Science.* Mathematisch Centrum, 1975.

3. Faustini, A. *The Equivalence of an Operational and a Denotational Semantics for Pure Dataflow.* Ph.D. Th., University of Warwick, 1982.

4. Ginzburg, S. and H. Rice. "Two Languages Similar to ALGOL-60". *Communications of the ACM 2* (1962).

5. Kahn, G. The Semantics of a Simple Language for Parallel Programming. Information Processing 74: Proceeding of the IFIP Congress 74, 1974, pp. 471-475.

6. Manna, Z.. *Mathematical Theory of Computation.* McGraw-Hill Publishing Company, New York, 1981.

7. Tarski, A. "A Lattice-theoretic Fixpoint Theorem and its Applications". *Pacific Journal of Mathematics ?* (1955).