Technical Report 633

Foundations of Actor Semantics

William Douglas Clinger

MIT Artificial Intelligence Laboratory

Foundations of Actor Semantics

by

William Douglas Clinger

May 1981

This report reproduces, with corrections, a dissertation submitted 1 May 1981 to the Massachusetts Institute of Technology Department of Mathematics in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Foundations of Actor Semantics

by

William D Clinger

Submitted to the Department of Mathematics
on 1 May 1981 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy in
Mathematics and Computer Science

Abstract

The actor message-passing model of concurrent computation has inspired new ideas in the areas of knowledge-based systems, programming languages and their semantics, and computer systems architecture. The model itself grew out of computer languages such as Planner, Smalltalk, and Simula, and out of the use of continuations to interpret imperative constructs within λ-calculus. The mathematical content of the model has been developed by Carl Hewitt, Irene Greif, Henry Baker, and Giuseppe Attardi. This thesis extends and unifies their work through the following observations.

The ordering laws postulated by Hewitt and Baker can be proved using a notion of global time. The most general ordering laws are in fact equivalent to an axiom of realizability in global time. Independence results suggest that some notion of global time is essential to any model of concurrent computation.

Since nondeterministic concurrency is more fundamental than deterministic sequential computation, there may be no need to take fixed points in the underlying domain of a power domain. Power domains built from incomplete domains can solve the problem of providing a fixed point semantics for a class of nondeterministic programming languages in which a fair merge can be written.

The event diagrams of Greif's behavioral semantics, augmented by Baker's pending events, form an incomplete domain. Its power domain is the semantic domain in which programs written in actor-based languages are assigned meanings. This denotational semantics is compatible with behavioral semantics.

The locality laws postulated by Hewitt and Baker may be proved for the semantics of an actor-based language. Altering the semantics slightly can falsify the locality laws. The locality laws thus constrain what counts as an actor semantics.

Thesis Supervisor: Dr Carl Hewitt
Title: Associate Professor of Electrical Engineering and Computer Science

This thesis is dedicated to
Richard Browning and Bruce Acklin.

Acknowledgements

I wish to thank:

Carl Hewitt, my thesis advisor, for years of patient help, and for his vision of computing, the actor model of computation.

Robert Halstead and Michael Sipser, my readers and members of my interdepartmental committee, for their willingness to read and question a thesis that lately threatened to disappear into a chasm betwixt their respective departments.

Giuseppe Attardi and Irene Greif, for sharing their ideas and suggesting new directions.

Giuseppe Attardi, Bob Berwick, Dean Brock, Sy Friedman, Anne Hartheimer, William Kornfeld, Bruce Schatz, Richard Stallman, Gerald Sussman, and Barbara White, for reading and remarking upon early drafts of pieces of this thesis.

The members of the MIT Artificial Intelligence Laboratory, for ideas and encouragement, friendship and tutelage.

The citizens of these United States, for funding the first three years of my graduate education through a National Science Foundation Graduate Fellowship in mathematics. May their generosity redound to the benefit of all nations.

Contents

Chapter I Introduction 8

1. Fairness 9
2. Overview 12
3. Related Research 14

Chapter II Ordering Laws 16

1. The Actor Model 16
2. Global Time is Necessary 21
3. A Mathematical Formulation 23
4. Time, Causality, and Computation 26
5. The Strong Axiom of Realizability 30
6. The Weak Axiom of Realizability 37
7. A Strong Independence Result 42
8. Modifying a Proof 46

Chapter III Nondeterminism 52

1. Nondeterminism can be Viewed as Incomplete Specification 52

2. Fixed Point Semantics 58
3. Domains and Their Completions 60
4. The Power Domain 68
5. Power Domains from Incomplete Domains 75
6. Implementations are not Meanings 78
7. Choice Nondeterminism is Bounded 79
8. Fairness Implies Unbounded Nondeterminism 82

Chapter IV Actor Semantics 87

1. Primitive Serializers 87
2. Actor Behaviors 89
3. The Actor Event Diagram Domain 91
4. Meanings as Fixed Points 99
5. Example: Infinite Loop 108
6. Example: Terminating Unbounded Choice 111
7. Example: Possibly Nonterminating Choice 116
8. Relation to Standard Power Domains 120

Chapter V Locality Laws 123

1. Actor Acquaintances 123
2. Actor Creation 124
3. Locality Laws Add Power 128
4. Semantics with Actor Creation 132
5. A Toy Language 136
6. The Locality Laws may not Hold 137
7. The Locality Laws may be Provable 138

Chapter VI Conclusion 141

Appendix I Atolia: Informal Description 143

Appendix II Atolia: Sample Programs 153

Appendix III Atolia: Comparison with Act1 and CSP 158

Appendix IV Atolia: Formal Semantics 161

References 174

Chapter I

Introduction

Today's algorithmic programming languages were designed to express deterministic sequential algorithms. They were not designed to express algorithms for the distributed computer networks and network-like multiprocessors that are now being designed and built. Algorithms for these networks and multiprocessors make use of concurrent computation and are often nondeterministic in that they do not specify a unique outcome.

The now classic Scott-Strachey theory of programming language semantics deals only with deterministic programming languages. That is, using the Scott-Strachey theory to describe the semantics of a language defines a unique mathematical object for every well-formed language construct. The hallmark of nondeterministic programming languages, however, is a kind of semantic ambiguity: some programs may for a given input produce any of several possible outputs.

Why not extend the Scott-Strachey theory by making the mathematical object corresponding to the output of a nondeterministic program be the set of its possible outputs? Gordon Plotkin has done precisely that in working out the theory of power domains, so called by analogy with

power sets.[1] Each element of a power domain is a set of possible outcomes of a nondeterministic program or program fragment. One of the most important shortcomings of power domains has been their seeming inability to deal with fair merge, finite delay, unbounded nondeterminism, and other manifestations of fair parallelism.

This thesis presents a theory of semantics for a class of nondeterministic programming languages with fair parallelism. Specifically, this thesis is concerned with programming languages based on the actor model of concurrent computation.[2] Actor semantics shows that power domains can be made to overcome the problem of fairness.

1.1. Fairness

Consider the problem of scheduling disk operations requested by concurrent processes. Because the disk is slow relative to the processes, requests should be buffered; let's call a request in the buffer a pending request. Two possible scheduling strategies are the First Come First Served strategy and the Shortest Seek Time First strategy. The First Come First Served strategy services pending requests in the order they arrive at the scheduler. The Shortest Seek Time First strategy attempts to minimize disk head motion by always servicing the pending request that involves moving the disk head the shortest distance. In many cases the Shortest Seek Time First strategy gives better average response time than the First Come First Served strategy.[3] Unfortunately, the Shortest Seek Time First strategy is incorrect because it cannot guarantee that every pending request will be serviced.

Figure 1 shows why. Process P_0 wishes to read a cylinder near the center of the disk. Process P_1 wishes to read and write cylinders near the disk's outer edge. The disk head happens to be over P_1's cylinders. Suppose process P_1, in a burst of activity, sends fifty or so requests to the disk scheduler, all involving cylinders near the outer edge of the disk. Suppose furthermore than whenever process P_1 receives confirmation that one of its requests has been serviced, it sends yet another request to the disk

[1]G D Plotkin, "A powerdomain construction", *SIAM J Computing* 5, 3, September 1976, pages 452–487.

[2]For a very diverse, nontechnical, amusing introduction to actor-based languages, see Ted Nelson [editor], "Symposium on actor languages", *Creative Computing* 6, 10, October 1980, pages 61–86, continued in *Creative Computing* 6, 11, November 1980, pages 74–94.

[3]Micha Hofri, "Disk scheduling: FCFS vs. SSTF revisited", *CACM* 23, 11, November 1980, pages 645–653. This article fails to observe that without modification the SSTF algorithm is incorrect.

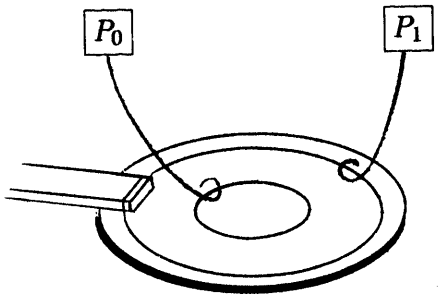


Figure 1. Disk cylinders accessed by two processes.

scheduler. If the disk scheduler is using the Shortest Seek Time First strategy, process P_1 will capture the disk. Process P_0 will be locked out, and any disk requests made by process P_0 will remain pending forever. That isn't fair.

On the other hand the First Come First Served scheduling strategy is fair. For that very reason, however, it causes problems for power domain semantics. For example suppose process P_0 makes one disk request, represented by a 0, while process P_1 makes infinitely many disk requests, each represented by a 1. This situation is diagrammed in Figure 2, where 1^ω indicates an infinite sequence of ones.[4] As is usual in programming language semantics, time has been left out of the picture in order to obtain a more abstract description—but as a result it is impossible to say where the 0 should appear in the output of the First Come First Served scheduler. Depending on the timing, the output could be any of

[4]Throughout this thesis ω is the first infinite ordinal, the first infinite cardinal, and the set of natural numbers with the usual ordering. Identifying these three conceptually distinct objects is a vice common among mathematicians who have studied set theory.

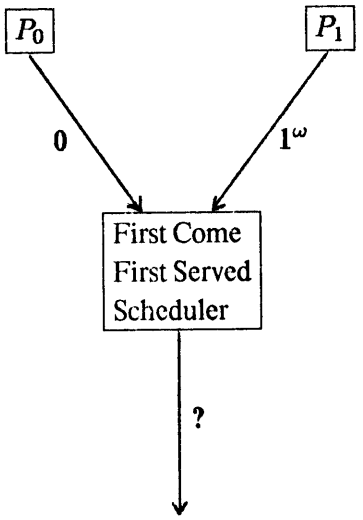


Figure 2. Data flow diagram of a scheduling problem.

$$011111111\cdots$$
$$101111111\cdots$$
$$110111111\cdots$$
$$111011111\cdots$$

and so on. The infinite sequence of ones is not a possible output, though, because it does not contain the 0 that is sent to the scheduler by process P_0. In other words the First Come First Served scheduler, abstractly considered, performs an arbitrary fair merge on its inputs.

Notice that nondeterminism is a property of our abstract description of the First Come First Served scheduling algorithm, not a property of the algorithm itself.

Nonetheless conventional power domain semantics attempts to account for nondeterminism in terms of choice points within the program's execution sequence. In the case of a merge program the choice points represent decisions about which value to output next. Figure 3 shows the choice tree for a merge of 0 and 1^ω. At the beginning of execution no outputs have been produced, so the root of the choice tree is labelled by a special symbol $\perp$ standing for the empty output. The program must then choose whether to produce 0 or 1 as its next output. If it produces 0, each subsequent output must be a 1. If it produces 1, however, it faces the same choice all over again.

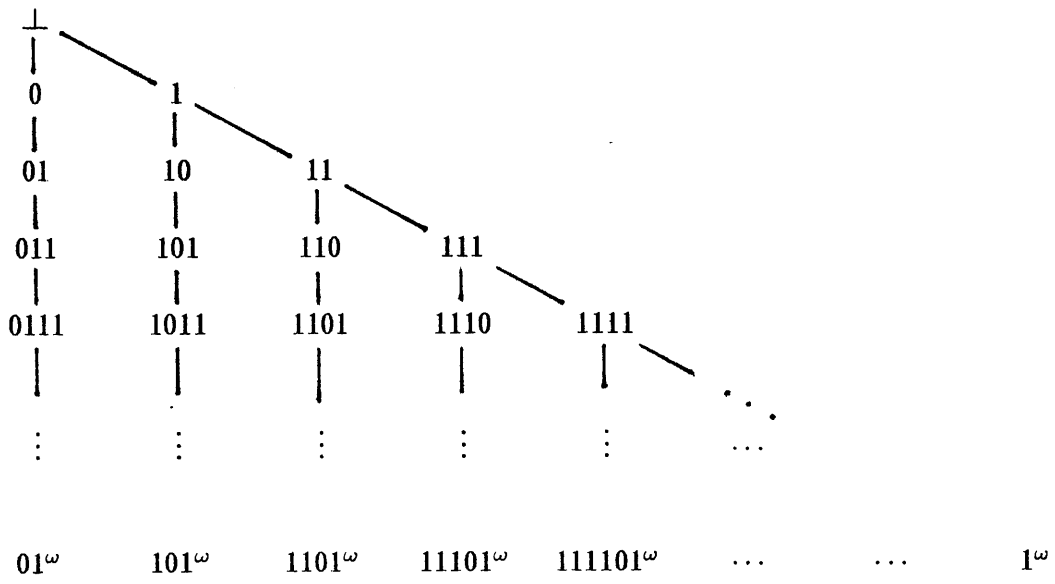


Figure 3. The choice tree for a merge of 0 and 1^ω.

Conventional power domain semantics regards each branch of the choice tree as a possible execution sequence of the merge. The possible outputs of the merge are the limits of these branches. Observe, however, that 1^ω is the limit of the rightmost branch, so the choice tree in Figure 3 does not represent a fair merge of 0 and 1^ω. In fact, no such choice tree drawn according to the rules of conventional power domain semantics can represent the arbitrary fair merge of 0 and 1^ω. As a corollary, conventional power domain semantics cannot give the abstract semantics of a First Come First Served scheduler.

Fair scheduling can be programmed in languages based on the actor model of computation.[5] Conventional power domains are therefore inadequate as a basis for actor semantics. Chapters III and IV develop and illustrate unconventional power domains that can deal with fair parallelism.

[5]Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying and proving properties of guardians for distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Notes in Computer Science 70, 1979.

1.2. Overview

The actor message-passing model of concurrent computation has inspired new ideas in the areas of knowledge-based systems,[6] programming languages and their semantics,[7] and computer systems architecture.[8] The model itself grew out of computer languages such as Planner,[9] Smalltalk,[10] and Simula,[11] and out of the use of continuations to interpret imperative constructs within λ-calculus.[12] The mathematical content of the model has been developed by Carl Hewitt, Irene Greif,[13] Henry Baker,[14] and Giuseppe Attardi.[15] This thesis extends and unifies their work.

Chapter II introduces the actor model and gives a mathematical definition of the actor event diagrams introduced by Greif.[16] The main result of Chapter II is that the most general ordering laws postulated by Hewitt and Baker[17] are equivalent to an axiom of realizability in global time. A strong independence result further emphasizes the importance of global time in the actor model, and suggests that some notion of global time is essential to any model of concurrent computation.

Chapter III discusses nondeterminism. It argues that nondeterminism in a programming language semantics is better understood as incomplete specification than as random choice. It follows

[6]E.g. Kenneth M Kahn, "An actor-based animation language", *Creative Computing* 6, 11, November 1980, pages 75–84.

[7]E.g. Guy Lewis Steele Jr and Gerald Jay Sussman, "Scheme: an interpreter for extended lambda calculus", MIT AI Memo 349, December 1975.

[8]E.g. the design of the Intel 432 was influenced by the Actor model.

[9]Carl Hewitt, "Planner: a language for manipulating models and proving theorems in a robot", *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington DC, 1969.

[10]Daniel H H Ingals, "The Smalltalk-76 programming system: design and implementation", *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson AZ, January 1978, pages 9–16.

[11]W R Frantz, "Simula language summary", and Kristen Nygaard and Ole-Johan Dahl, "The development of the Simula language", two papers presented at the ACM SIGPLAN History of Programming Languages Conference, Los Angeles CA, June 1978, preprint in *SIGPLAN Notices* 13, 8, August 1978, pages 243–272.

[12]E.g. Michael J C Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

[13]"Semantics of communicating parallel processes", MIT Project MAC Technical Report 154, September 1975.

[14]"Actor systems for real-time computation", MIT LCS Technical Report 197, March 1978.

[15]Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying and proving properties of guardians for distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Notes in Computer Science 70, 1979.

[16]"Semantics of communicating parallel processes".

[17]"Laws for communicating parallel processes", IFIP-77, Toronto, August 1977, pages 987–992. "Actors and continuous functionals", IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, 16.1–16.21.

that the nondeterminism in a programming language semantics is, in David Park's term, loose nondeterminism.[18] The importance of these philosophical distinctions is that fairness implies unbounded nondeterminism, whereas viewing nondeterminism as random choice leads to the conclusion that all nondeterminism is bounded.

What is new in Chapter III is the treatment of power domains. Instead of beginning with a semantics for sequential programs and then trying to extend it for nondeterministic concurrency, actor semantics views nondeterministic concurrency as primary and obtains the semantics of sequential programs as a special case. The mathematical import of this approach is that there is no longer any need to take fixed points in the domain underlying a power domain. As a result the underlying domain need not be complete. Extending the power domain construction as in Chapter III to apply to incomplete domains makes possible a power domain semantics for a class of nondeterministic programming languages in which a fair merge can be written.

Chapter IV verifies that claim by presenting a specific power domain semantics for actor-based languages. The event diagrams of Greif's behavioral semantics, when augmented by Baker's pending events,[19] form an incomplete domain. Its power domain is the semantic domain in which programs written in actor-based languages are assigned meanings.

Chapter V points out that whether or not the locality laws postulated by Hewitt and Baker[20] hold for a toy language depends upon details of the language's semantic equations. The conclusion there drawn is that the locality laws constitute the acid test of a programming language's faithfulness to the actor model. Chapter V also extends the semantics of Chapter IV to deal with actor creation.

The concluding chapter, Chapter VI, suggests some directions for further research.

The appendixes present the toy language used throughout the thesis to illustrate actors.

I.3. Related Research

Plotkin's original power domain construction was simplified by Michael Smyth, whose paper

[18]David Park, "On the semantics of fair parallelism", University of Warwick Theory of Computation Report 31, October 1979.

[19]"Actor systems for real-time computation".

[20]"Laws for communicating parallel processes" and "Actors and continuous functionals".

remains the standard introduction to the subject.[21] A number of nondeterministic programming languages have now been given a power domain semantics. Of these, the semantics of Communicating Sequential Processes[22] has had the most influence on actor semantics.

The semantics in Chapter IV is probably the first power domain semantics for languages with fair parallelism, but it is not the first power domain semantics to deal with unbounded nondeterminism. R J Back has given a power domain semantics for a language with unboundedly nondeterministic assignment statements as basic operations.[23] Three differences between Back's work and actor semantics stand out. One difference is the source of nondeterminism—basic assignment statements in Back's paper, message delays in actor semantics. A second difference is that Back is thinking of nondeterministic sequential programming languages, while actor semantics is concerned primarily with concurrent programming languages. The third difference is that Back's power domain apparently is constructed from a complete underlying domain. This third difference is not entirely clear because Back's power domain construction appears to be nonstandard. A similarity between Back's work and actor semantics is that Back found it necessary to build the power domain out of execution sequences instead of single states: the actor power domain is built out of actor event diagrams, which may be thought of as generalized execution sequences.

[21]"Power domains", *J Computer and System Sciences* **16**, 1978, pages 23–36.

[22]Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, December 1979, pages 290–308.

[23]"Semantics of unbounded nondeterminism", Mathematisch Centrum Report IW 135/80, April 1980.

Chapter II

Ordering Laws

This chapter illustrates the actor model at its most abstract. A notion of global time is introduced and used to prove the ordering laws postulated by Hewitt and Baker. Needlessly restrictive ordering laws are avoided, so that axioms of realizability in global time can be shown equivalent to the ordering laws. The importance of global phenomena is emphasized through a strong independence result. Finally, a theorem by Hewitt and Baker is shown to remain true under laws equivalent to a weak axiom of global time realizability.

II.1. The Actor Model

Ordinary sequential computation is the simplest case of concurrent computation, a far more general category that includes various kinds of parallel computation as well as the sequential case. While the sequential case is fairly well understood, however, general concurrent computation is not. There are two evident ways to develop a better theory of concurrent computation. One is to generalize the existing theory of sequential computation. The other is to begin with a model of concurrent computation and create an entirely new theory that can be checked against current theory in the special

case of sequential computation. The actor model is intended to support this second sort of theoretical development. Generalizing existing theory, as in the first approach, can lead to significantly different theoretical predictions, as will appear in Chapter III.

As a model of concurrent computation, the actor model emphasizes the communication occurring during computation. Examples of such communication are the signals transferred along the bus linking the CPU and memory of a conventional sequential computer, parameter passing between subroutines of a program, messages transferred between computers in a geographically distributed network, and process synchronization in a multiprocessing computer. All these communications may be considered examples of what has come to be called *message passing.*

The actor model is one of a number of message passing models that have been developed in the past decade.[1] These models differ in their conception of message passing. For some, the mechanism of message passing resembles a telephone network, so that message transmission is essentially instantaneous, but there are times when the line is busy and messages cannot be sent.[2] For the actor model, however, message passing resembles mail service, so that messages may always be sent but are subject to variable delays en route to their destinations. As a result, the actor model can be used to analyze distributed computer networks as well as multiprocessors and programs.

In the actor model, each communication is described as a *message* arriving at a computational agent called an *actor.* Memory chips, subprograms, and entire computers are examples of things that may be thought of as actors. The memory chip might receive addresses and function codes as messages, while the subprograms might receive values or locations of parameters, and the computer might receive messages as blocks or packets. The actor model refers to the arrival of a message at an actor as an *event.* Thus all events in the model are arrival events, and there is no such thing as a sending event.

The graphic representation of an event is a dot, as below.

[1]Two examples are C A R Hoare, "Communicating sequential processes", *CACM* **21**, 8, August 1978, pages 666–677, and George Milne and Robin Milner, "Concurrent processes and their syntax", *JACM* **26**, 2, April 1979, pages 302–321.

[2]This is one way to understand the semantics of "Communicating sequential processes". See Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, December 1979, pages 290–308.

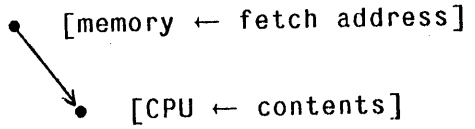
The actor that receives a message in an event is called the *target* of the event. The message that the target receives is just called the message of the event. The target and message of an event are often described by the notation

$$[target \leftarrow message]$$

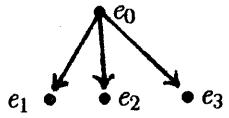
which may appear beside dots representing events.

Sometimes the target of an event, as a direct result of that event, will send messages to other actors. For example, a memory module receiving a message instructing it to fetch the contents of a certain address should respond by sending the value stored at that address to the CPU. In this case, the event of which the memory module is the target *activates* the event in which the contents of the specified address arrives at the CPU.

The activation relation appears as an arrow in diagrams.



An event may activate several subsequent events. That is, the arrival of a message at an actor may cause that actor to send out a number of messages to other actors. The events that a given event activates are said to have that event as their *activator.*

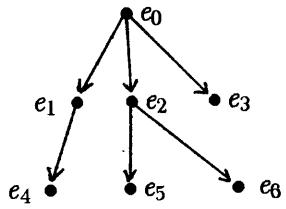


Thus e_0 activates e_1, e_2, and e_3, each of which has e_0 as activator. e_0 is an example of an *external event*, that is, an event with no activator. Its cause must be external to the system being modelled, hence the name.[3] No event has more than one activator, because the message of an event has been

[3]External events were called initial events in Carl Hewitt and Henry Baker, "Actors and continuous functionals", IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, 16.1–16.21. This usage conflicts with that in Carl Hewitt and Henry Baker, "Laws for communicating parallel processes", IFIP-77, Toronto, August 1977, pages 987–992. The usage of the latter paper is better motivated, since it defines an initial event as an event that is initial in the activation ordering considered as a category.

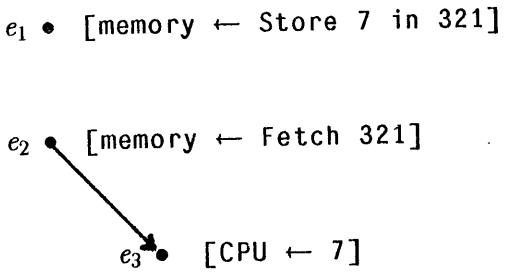
sent only once.

Chains of activations define the *activation ordering.*

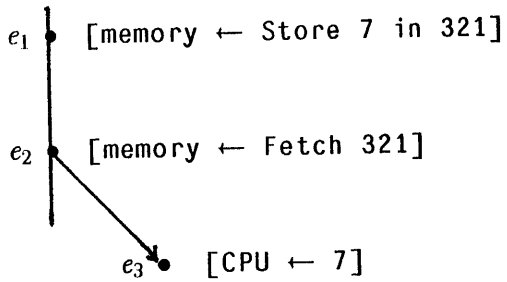


Thus e_0 activates e_2 which activates e_6, so e_0 precedes e_6 in the activation ordering. Similarly both e_0 and e_1 precede e_4 in the activation ordering. e_4 and e_3 are not related by the activation ordering.

Sometimes an event will not activate any other events. When that happens, the only effect of the event is whatever effect it may have on the (local) state of its target. Considering the memory module again, the message `Store 7 in 321` will probably cause it to change its state. In this way events can influence future events even though they do no activate any events themselves. Graphically

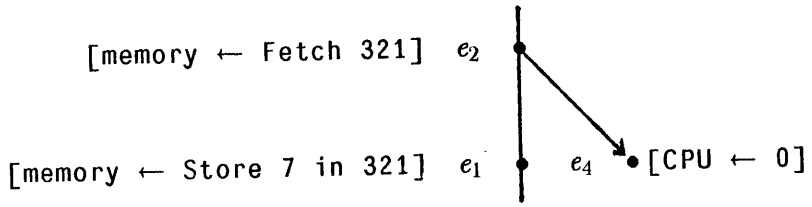


There is no explicit path in this diagram to show that e_3 depends upon e_1. To remedy that the actor model introduces the *arrival ordering* of the memory module, which appears as a vertical line.



Adding this arrival ordering shows that e_1 precedes e_3 in the *combined ordering*, which is simply the combination of the activation and arrival orderings.

The arrival ordering emphasizes that the relative order of e_1 and e_2 is significant.



Here e_1 does not precede e_4 in the combined ordering.

The actor model postulates an arrival ordering for each actor. These arrival orderings are supposed to be linear, which means that for any two events with the same target, it is always the case that one of the two occurs first. Some form of arbitration may be necessary to make this supposition realistic, of course.

An arrival ordering represents the order in which events occur at a particular target actor. Thus an arrival ordering represents the *local time* of an actor.

Conventional models of sequential computation make use of *global time* and *global state*. That is, there is only one clock in the system, and the computation is in exactly one well-defined state at any given time. The transitions between global states are linearly ordered in the global time of the system, which is what makes sequential computation sequential.

When computation is not sequential, the notions of global state and global time may be inappropriate. An extreme example suggests why. Suppose a computer in Dallas and another one in Oklahoma City are linked together to function as a dual processor. The computers are one millisecond apart at light speed. It is therefore not helpful to insist that events occurring with megahertz frequencies at the two sites must be thought of as totally ordered in a single global time, for an event in Dallas clearly cannot affect any part of a hypothetical global state on which an event nanoseconds later in Oklahoma City depends. Such concurrent systems are better analyzed by splitting the global state into local pieces and viewing the overall computation as a set of local computations interacting through message passing.

This kind of local decomposition is important for multiprocessor systems as well as for geographically distributed systems. Several experimental multiprocessors resemble computer networks, and

multiprocessor networks are becoming available commercially as well.[4]

Even large sequential programs are constructed from local modules that communicate through the conventional mechanisms of subprogram calls with parameters and shared variables. These mechanisms may also be regarded as special cases of message passing.[5]

The actor model emphasizes the ideas of local time and local state. Local times are represented by the arrival orderings of actors, which operate independently of each other except when they interact by means of message passing. The communications between actors are represented by the activation ordering. Hence the combined ordering indicates all possible dependencies among events. Since in the actor model events cannot be influenced by events that do not precede them in the combined ordering, the actor model helps to illustrate the modular structure of a computation. On the other hand, using a single global time to order computation events linearly makes it appear that an event depends upon all events that happen to come before it in global time.

II.2. Global Time is Necessary

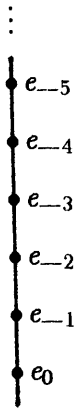
Nonetheless it turns out that some notion of global time is essential to any model of concurrent computation. The purpose of this chapter is to show why that is so for the actor model, and to use the idea of global time to motivate and improve upon the *ordering laws* introduced by Hewitt and Baker.[6]

So far the arrival orderings have been required only to be total. Consider, however, an arrival ordering with the same order type as the nonpositive integers.

[4]A commercial example I happen to be familiar with is the Advanced Flexible Processor built by the Information Sciences Division of Control Data Corporation. Up to sixteen of these processors can be configured in a simple bidirectional ring network, providing a computation rate of well over a billion fixed point arithmetic operations per second in some signal processing applications.

[5]Carl Hewitt, "Viewing control structure as patterns of passing messages", *Artificial Intelligence* 8, 1977, pages 323-363. Also in Winston and Brown [ed], Artificial Intelligence: an MIT Perspective, MIT Press, 1979.

[6]Carl Hewitt and Henry Baker, "Laws for communicating parallel processes", IFIP-77, Toronto, August 1977, pages 987-992. Carl Hewitt and Henry Baker, "Actors and continuous functionals", IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, 16.1–16.21.



This arrival ordering seems unlikely to arise in practice. For an even unlikelier arrival ordering, considerably harder to draw, consider the order type of the nonnegative rationals. These examples suggest that the actor model should place further restrictions on the arrival orderings. Such restrictions are stated by the ordering laws.

For example, one ordering law states that for any two events having the same target there are only finitely many events lying between them in the arrival ordering of the target. This law rules out arrival orderings having the order type of the nonnegative rationals, but does not rule out arrival orderings having the order type of the nonpositive integers. Another ordering law must be added to eliminate that order type. Other ordering laws must be stated to govern the activation ordering. To rule out the possibility of impossible situations arising from the interaction of allowable activation and arrival orderings, ordering laws must be stated for the combined ordering.

While laws can be generated by thinking of arrival, activation, and combined orderings having undesirable order types and then postulating ordering laws that eliminate them, it would never be possible to have total confidence that all undesirable order types have been ruled out by such a process. In other words, this ad hoc approach leaves open the question of the *sufficiency* of the ordering laws. Another, less important question concerns the *independence* of the laws. For example, Hewitt and Baker conjectured that their law governing the combined ordering was redundant, but could not prove it.[7]

The questions of independence and sufficiency turn out to be related, in that the question of in-

[7] "Actors and continuous functionals".

dependence points to the importance of global time, which provides an intuitive basis for considering the question of sufficiency.

The answer to the question of sufficiency runs as follows. The ordering laws are nothing more than conditions necessary for orderings to be realizable in global time. They should therefore be considered complete if they form a necessary and sufficient set of conditions for orderings to be embedded in global time. The three strongest ordering laws form such a complete set. That is the message of Theorem 1 of §5, wherein they are shown equivalent to a statement of global time realizability.

As for the question of independence, the three strongest ordering laws are strictly stronger than the conjunction of all the other ordering laws, even in the presence of the *locality laws*[8] discussed in Chapter IV. In particular, the law governing the combined ordering is independent of the other laws, which explains why Hewitt and Baker were unsuccessful in proving their conjecture. The reason for this law's independence is that the combined ordering is a global ordering, while the other laws deal only with local orderings, namely the activation and arrival orderings. As shown by this law's independence, local laws are not by themselves enough. A global law is needed to make the actor model an adequate account of concurrent computation.

II.3. A Mathematical Formulation

So far the actor model has been described informally. A more rigorous presentation at this point will avoid some confusion later on, as well as provide a chance to review the model. Some details of the actor model, such as the contents of messages and the behaviors of actors, make no difference when discussing the ordering laws. Hence they will not be discussed now, but will reappear later. The simplified actor model used in this chapter is less detailed and more general than the versions considered in chapters III, IV, and V.

The actor model is perhaps best motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds, or even thousands of independent monoprocessors, each with its own

[8]Hewitt and Baker, "Laws for communicating parallel processes" and "Actors and continuous functionals".

local memory and communications processor, communicating via a high performance communications network in a system much like the computer networks now coming into widespread use. The model may be thought of as an idealization of such a multiprocessor network, in which the number of available processors is potentially infinite, much as the tape of a Turing machine is potentially infinite.

The primitive objects of the simplified model are events and actors. The actors represent computational agents. In the idealization suggested above, an actor may be thought of as a program that has been given its very own processor on which to run. An event represents the arrival of a message at a target actor.

The model uses partial orders on these events to represent concurrency. There is a treelike activation ordering that represents causality, and a set of linear arrival orderings, one for each actor, that represent local times. The combined ordering is the transitive closure of the activation and arrival orderings, and may be considered to represent feasible concurrency. The combined ordering is similar to the concurrency orderings of some other models, but its decomposition into activation and arrival orderings is unique to the actor model.

Write the set of events of a computation as E, and the set of actors as A. Associated with each event is its target actor, so let T:E $\rightarrow$ A be the function giving the target of each event. The model does not need to record the sender as well as the target, because the sender can be determined from the activation ordering unless the event is external. The events with a given target are linearly ordered by the arrival ordering of the target, so let Arr be a collection of irreflexive total orderings $-arr_a\rightarrow$ defined on $T^{-1}(a)$, for $a \in$ A. There is also the activation ordering $-act\rightarrow$, an irreflexive partial order on E such that no event has more than one immediate predecessor.[9]

A computation thus becomes a structure

$$\langle \text{E}, \text{A}, T, -act\rightarrow, \text{Arr}\rangle.$$

Not all such structures correspond to reasonable computations, however. The purpose of the ordering laws is to characterize those structures that represent real computations.

[9] x is an immediate predecessor of z with respect to an irreflexive ordering $<$ if $x < z$ but there is no y such that $x < y < z$.

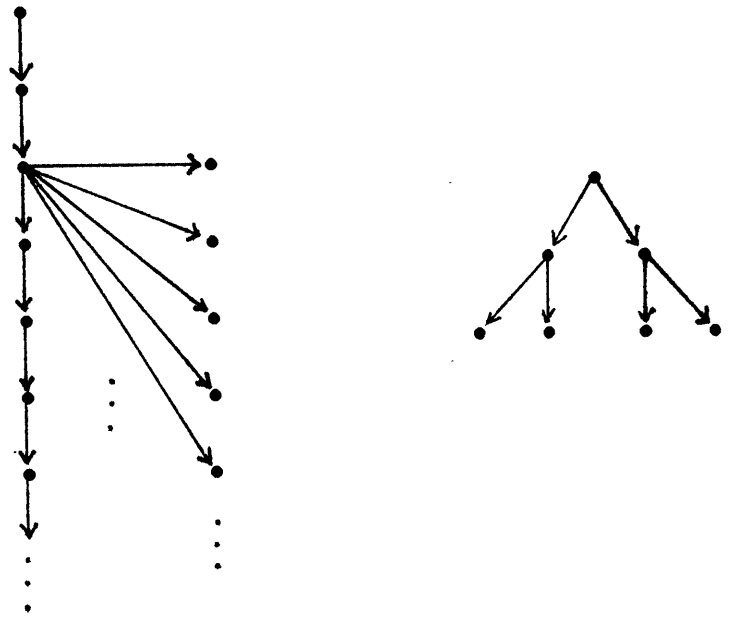


Figure 1. An example of an activation ordering with two components.

Some readers may be uncomfortable with the infinities allowed by such a structure. The considerations of the next section will require that the set of events E be countable. E cannot be required to be finite because that would make the model useless for nonterminating computations. For the same reason there may be infinitely many external events, which are simply events having no predecessors in the activation ordering. External events are intended to represent events whose cause is external to the system being modelled, such as the event of pressing a button or kicking the machine. There must be at least one such event in a nonempty computation, but there is no reason to insist that there be only one. Each external event defines a component of the activation ordering, and each component is a tree with the external event as its root. See Figure 1.

Figure 1 also illustrates the fact that an event can activate infinitely many events.[10] For example, receiving a message can cause an actor to enter an infinite loop in which it continues to send out mes-

[10]Hewitt and Baker did not allow this. §8 shows how to modify a proof of theirs that assumed that events can activate only finitely many events.

sages. Another example, motivated by the language Ether,[11] is an event that results in broadcasting a message to every present and future actor.

The number of actors must be potentially infinite because at times actors represent software entities such as programs and functions, and in languages such as Lisp new functions can be generated automatically and endlessly.

II.4. Time, Causality, and Computation

Let $\rightarrow$ denote the combined ordering, which is the transitive closure of the activation ordering $-act\rightarrow$ and the arrival orderings in **Arr**. If an event e_1 precedes another event e_2 in the combined ordering, then there exists a path of causation and local time from e_1 to e_2. If that is so then e_1 must occur before e_2 in time. It makes no difference whether time is measured in the reference frame of the target of e_1, the target of e_2, or in any other reference frame, for the existence of the path of causation and local time between e_1 and e_2 implies that the time sequence of the two events is invariant among all observers. Some time relations are absolute, even in the theory of relativity.

Pursuing that thought a bit further, the theory of relativity allows each observer his or her own global time. These global times may differ, however, concerning the order of events whose relation in time is not absolute.

There is an analogy with global time in the actor model. When e_1 precedes e_2 in the combined ordering, all global times must have e_1 happening before e_2. When e_1 and e_2 are not comparable under the combined ordering, however, there will be global times in which e_1 happens first and other global times in which e_2 happens first.

The mathematical notion of global time appropriate for event-structured models of computation is of a function from the computation events into the real numbers. Often the global time function is required to be integer-valued, and that will turn out to be the case for the actor model, but for now it will just be a real-valued function. For the actor model, then, a global time is a mapping

$$g: \mathrm{E} \rightarrow \Re$$

[11] Bill Kornfeld, "ETHER—a parallel problem solving system", IJCAI-79, pages 490–492.

Figure 2. A combined ordering that is not irreflexive. (Arrival orderings point downward.)

where $\Re$ denotes the real numbers.

The reason for considering global times is that commonly held notions about time and computation will constrain the structures possible for the combined ordering and thus allow an intuitive derivation of the ordering laws.

One constraint on the global time mapping g is that cause precedes effect. Thus

[1] g preserves the activation ordering $\text{—}act\rightarrow$.

That is, if $e_1 \text{—}act\rightarrow e_2$, then $g(e_1) < g(e_2)$.

Another constraint is that global time be consistent with all local times. Thus

[2] g preserves all the arrival orderings $\text{—}arr_a\rightarrow$, for $a \in \mathrm{A}$.

Consequently

[3] g preserves the combined ordering $\rightarrow$.

and

[4] The combined ordering $\rightarrow$ is irreflexive.

[3] and [4] are equivalent to [1] and [2]. Irreflexivity of the combined ordering does not follow from irreflexivity of the activation and arrival orderings, as illustrated in Figure 2. It must be stated as a fundamental ordering law. Hewitt and Baker named it the Law of Strict Causality.

Law of Strict Causality. *The combined ordering $\rightarrow$ is an irreflexive partial ordering.*

So-called Zeno machines are paradoxical machines that can do infinitely many things in a finite amount of time. An example is Huffman's Lamp, which when switched on lights for only thirty seconds before turning itself off for fifteen seconds, and then comes back on for seven and a half seconds before turning off for three and three quarters seconds, and so on. After one minute it ceases to change state. At one second into the second minute, is it on or off? Zeno machines, if they existed, could be used for many useful purposes such as providing a decision procedure for first order predicate calculus. The fact that they do not exist leads to requiring that

[5] The range of g has no accumulation points.

Equivalently, no bounded interval in $\Re$ contains infinitely many images of E under g. Equivalently, because the combined ordering is irreflexive, a global time g can be found that is integer-valued and one-to-one.

Together with [5] above, the following implies that there is a first event, and thus that the computation has a definite beginning.

[6] the range of g is a subset of the nonnegative real numbers.

Putting the above constraints together yields the fundamental axiom on actor orderings, the

(Strong) Axiom of Realizability. *There exists a one-to-one mapping g from the events* E *into the nonnegative reals that preserves the combined ordering $\rightarrow$ and such that $g^{-1}(I)$ is finite for every bounded interval I of $\Re$. Equivalently there exists a one-to-one mapping g:E $\rightarrow \omega$ that preserves $\rightarrow$, where ω is the set of natural numbers.*

Occasionally there may be reason to weaken this axiom slightly by not assuming a definite beginning to the computation as in [6]. For example, many properties of a computer network that has been operating continuously for years will in no way depend upon there having been a time before the system was brought up, and so any proof that made use of that fact would be suspect. On the other hand, if the assumption really is necessary to the proof, then that tells something about the property being proved, namely that it depends upon the existence of some initial state. For these reasons, and against the chance that steady state theory may come back into fashion in cosmology, this chapter will also consider the

Weak Axiom of Realizability. *There exists a one-to-one mapping g from the events* E *into the real numbers* $\Re$ *that preserves the combined ordering* $\rightarrow$ *and such that* $G^{-1}(I)$ *is finite for every bounded interval I of* $\Re$. *Equivalently there exists a one-to-one mapping* g:E $\rightarrow$ **Z** *that preserves* $\rightarrow$, *where* **Z** *is the set of integers.*

As will be shown, the ordering laws follow from the definition of the structure

$$\langle \mathrm{E}, \mathrm{A}, T, -act\rightarrow, \mathrm{Arr} \rangle$$

together with one of the versions of the realizability axiom.

Two of the ordering laws stated by Hewitt and Baker do not so follow, however, and are not in fact true in the system of this chapter. One of the laws asserted the existence of an initial event preceding all other events in the activation ordering. This was nothing more than a simplifying assumption appearing only in the paper "Laws for communicating parallel processes". The other asserted that an event can activate only finitely many events. The previous section gave two examples to justify omitting this law, one of them being the possibility of an actor entering an infinite sending loop. Apparently Baker wished to rule out the possibility of loops internal to actors.[12] It is also possible that the choice of the phrase "immediate successors in the activation ordering", while well grounded in established mathematical usage, may have led to thinking of immediate in the sense of time rather than in the sense of being without intervening events.[13]

[12]Henry Baker, "Actor systems for real-time computation", MIT LCS Technical Report 197, March 1978, page 64.

[13]*ibid*, page 37.

II.5. The Strong Axiom of Realizability

An *actor event diagram* is defined to be a structure

$$\langle \mathrm{E}, \mathrm{A}, T, -act\rightarrow, \mathrm{Arr} \rangle$$

that satisfies the strong Axiom of Realizability, where E and A are arbitrary sets and T, $-act\rightarrow$, and Arr are as described in §3. This section considers the ordering laws as consequences of that definition, while the next section considers weaker ordering laws that still hold when the strong axiom is replaced by the Weak Axiom of Realizability.

The global time g whose existence is asserted by the axioms of realizability is not part of the structure of an actor event diagram. The axioms assert only that it is possible to embed the activation and arrival orderings in time in a certain way. Generally there are many acceptable embeddings. Thus, although a particular actor event diagram must be realizable in time, no time sequencing is associated with it except the combined ordering. Furthermore, as shown by the main theorems of this and the next section, the realizability axioms are equivalent to certain simple ordering laws, so that the set of actor event diagrams may be defined using the ordering laws instead of a realizability axiom, and the definition need never explicitly mention global times at all.

Apparently the global time itself is seldom needed in practice. The mere possibility of one is quite constraining, implying as it does the ordering laws, and the ordering laws are generally more convenient for proofs. It is usually easier to prove properties of computations by considering the partial orderings themselves than by considering all possible global times, since in considering all possible linearizations of the partial orders in global time the proof still has to rely on properties of the partial orders. Hence there is no point to disguising the partial orders by mapping them into linear time.

As an example, consider the parallelism fork and join in Figure 3. Here an actor executing a process sends messages to two other actors asking them to start subprocesses to be computed in parallel with the main process. Either subprocess may finish and return its result first, so Figure 3 shows two possibilities for the join. Each actor event diagram in Figure 3 can be embedded in time in essentially three ways. For the event diagram on the left, the order of events in global time must be

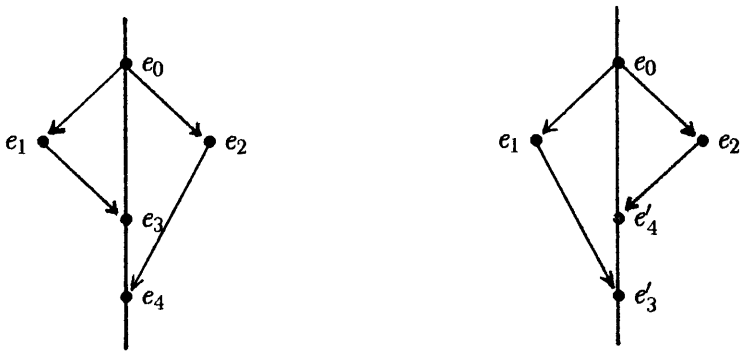


Figure 3. Parallelism fork and join.

one of

$$e_0, e_2, e_1, e_3, e_4$$
$$e_0, e_1, e_2, e_3, e_4$$
$$e_0, e_1, e_3, e_2, e_4$$

but it makes no difference which. Hence the additional ordering information given by the global time is useless. Since the global time disguises the fact that e_2 and e_3 cannot influence one another, the global time actually gives less information than the actor event diagram.

An exceptional situation when it is just as efficient to consider all global times arises when considering all interleavings of elementary operations in a multiprocessor system where communication is by means of shared memory.[14] In this instance the possible arrival orderings of the shared memory when considered as an actor are essentially the same as the possible interleavings, so there is nothing to gain from the actor point of view. In short, the local time of the memory is effectively the global time of the system. In less centralized, more modular systems, however, considering the partial orders directly is superior to considering their many linearizations. Once the ordering laws and their equivalence to the global time axioms have been derived, therefore, the realizability axioms will have fulfilled their main purpose.

Most logics that have been proposed for reasoning about parallel programs are based upon sequences of global states. The realizability axioms suggest that the actor model may be made com-

[14]See for example J M Cadiou and J J Levy, "Mechanizable proofs about parallel processes", Proceedings 14th Annual Symposium on Switching and Automata Theory, October 1973, pages 34–48.

patible with these logics by treating an event as a change of global state, so that a global time specifies a sequence of global states. To do so, however, is to sacrifice the advantages being claimed for the actor model. The actor model requires its own verification logic, which remains to be developed. The semantics presented in Chapter III may be used to justify the proof rules of such a logic.

The first two ordering laws follow from either the weak or the strong realizability axiom. They are the

Law of Strict Causality (LSC). *For no* $e \in \mathrm{E}$ *does* $e \rightarrow e$.

and the

Law of Countability (LC). *There are at most countably many events. That is,* E *is countable, where a finite set is considered countable.*

The first law was stated by Hewitt and Baker[15] and the second is provable in the system of "Laws for communicating parallel processes".

When the strong axiom is assumed, the intuition that events are only finitely removed from the beginning of computation comes back out as the

Law of Finite Predecession (LFP). *For all events* e_1 *the set* $\{ e \mid e \rightarrow e_1 \}$ *is finite.*

These three laws are in fact equivalent to the Strong Axiom of Realizability. It is thus a matter of choice whether to formalize actor event diagrams using the strong realizability axiom as has been done here or using these three ordering laws instead.

Theorem 1. *The strong Axiom of Realizability is equivalent to the conjunction of the Law of Strict Causality, the Law of Countability, and the Law of Finite Predecession.*[16]

Proof. The realizability axiom is easily seen to imply all three (LSC, LC, and LFP).

Let $\{ e_0, e_1, e_2, \ldots \}$ be the set of events. Define a global time g inductively as follows.

[15] "Laws for communicating parallel processes" and "Actors and continuous functionals".

[16] By assuming the existence of a single initial event that precedes all other events, and that no event can activate infinitely many events, Hewitt and Baker were able to prove that the Law of Discreteness (given in the next section) implied a statement equivalent to the strong Axiom of Realizability. Under their assumptions the Law of Countability and the Law of Finite Predecession also hold, so they had a greatly weakened version of the "if" part of this theorem. See §2.1 of "Laws for communicating parallel processes".

Let $g(e_0) = 1$.

Suppose that g has been defined on $\{ e_0, \ldots, e_{n-1} \}$ in such a way that it preserves the combined ordering $\rightarrow$ on the events on which it is defined. That is, $g(e_i) < g(e_j)$ whenever $e_i \rightarrow e_j$ for $i, j < n$. The strategy for defining $g(e_n)$ will be to place it as far to the right as possible. Precisely, if there exists a $j < n$ such that $e_n \rightarrow e_j$, then let k be such that

$$g(e_k) = \min\{ g(e_j) \mid e_n \rightarrow e_j, \quad j < n \}.$$

Define

$$g(e_n) = \frac{1}{2}\Big(g(e_k) + \max(\{ g(e_j) \mid g(e_j) < g(e_k), \quad j < n \} \bigcup \{0\}) \Big)$$

so that $g(e_k)$ is the first point on the right of $g(e_n)$. The claim is that g is now defined on $\{ e_0, \ldots, e_{n-1}, e_n \}$ in such a way as to preserve the combined ordering. If not, then, by the induction hypothesis and the fact that $g(e_n) < g(e_j)$ whenever $e_n \rightarrow e_j, \quad j < n$, there must be some particular $i < n$ such that $e_i \rightarrow e_n$ but $g(e_n) < g(e_i)$. This implies also that $g(e_k) < g(e_i)$. Now since $e_n \rightarrow e_k$, the transitivity of the combined ordering gives $e_i \rightarrow e_k$, which by LSC contradicts the fact that g preserves $\rightarrow$ on $\{ e_0, \ldots, e_{n-1} \}$. Thus no such i can exist, and g has been extended to $\{ e_0, \ldots, e_{n-1}, e_n \}$ while still preserving the combined ordering.

If there is no such $j < n$ such that $e_n \rightarrow e_j$, then just put $g(e_n)$ out to the right of all other points defined so far, say

$$g(e_n) = 1 + \max\{ g(e_j) \mid j < n \}.$$

As before, the combined ordering is preserved.

By induction the combined ordering is preserved at all stages. Any non-preservation of that ordering in the whole function g would already have arisen at some finite stage, and so g is a one-to-one positive-valued function that preserves the combined ordering. It only remains to be shown that its range has no limit points. This is equivalent to showing that the left-open unit intervals with integral endpoints, that is, intervals of the form $(m, m+1]$ for m a natural number, each contain only finitely many points of the range.

If $(m, m+1]$ contains any range points at all, then by the way g is defined $m + 1 = g(e_n)$ for some n, and the interval $(m, m+1]$ contains none of the points $g(e_0), \ldots, g(e_{n-1})$. That is, the

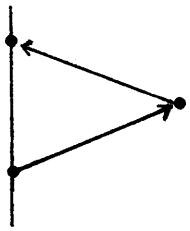


Figure 4. Irreflexive activation and arrival orderings do not imply an irreflexive combined ordering.

interval was empty when $g(e_n)$ was defined. Now it happens that the pre-images of all range points placed in that interval after $g(e_n)$ precede e_n in the combined ordering. Whenever $g(e)$ is defined to be a non-integer, e precedes the pre-image of the range point immediately to its right at the time of its definition. Thus the pre-image of the first range point placed in $(m, m + 1]$ after $g(e_n)$ precedes e_n in the combined ordering. The second does also, by transitivity of $\rightarrow$ if needed, and so on for all the range points placed in the interval. Hence if g takes infinitely many values in the interval $(m, m + 1]$ then there must be infinitely many events that precede e_n in the combined ordering. This contradicts the Law of Finite Predecession. ∎

The proof just given reveals that if $e, e' \in E$ are not related by the combined ordering, then there exists a global time g such that $g(e) < g(e')$.

The Law of Finite Predecession has two immediate corollaries concerning the primitive, local orderings, but taken together they remain weaker than LFP itself.

Law of Finite Predecession in the Activation Ordering. *For all events* e_1 *the set*

$$\{ e \mid e \text{ —}act\rightarrow e_1 \}$$

is finite.

Law of Finite Predecession in an Arrival Ordering. *For all events* e_1 *and actors* a *the set*

$$\{ e \mid e \text{ —}arr_a\rightarrow e_1 \}$$

is finite. (Of course the set is empty if $T(e_1) \neq a$.*)*

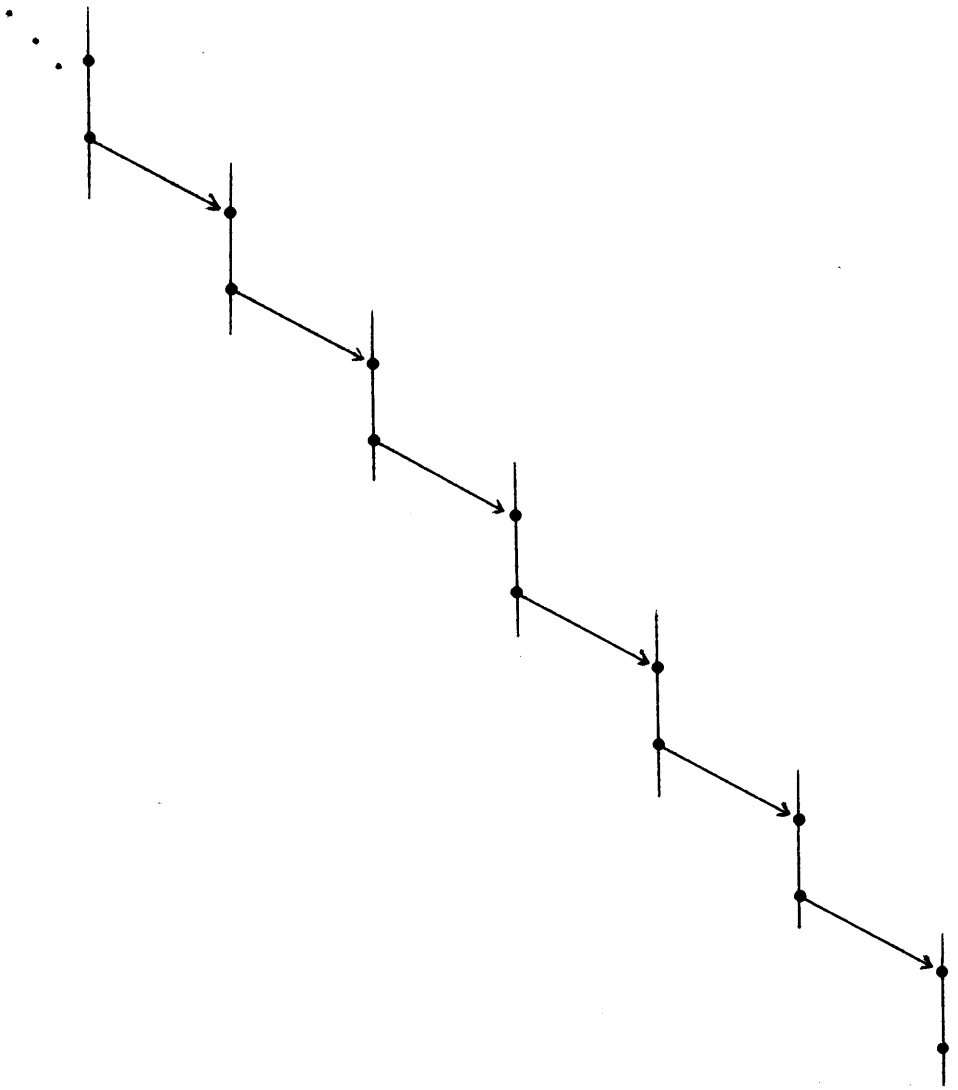


Figure 5. An infinite backward chain in the combined ordering.

Theorem 2. *The strong Axiom of Realizability is stronger than*

1. The conjunction of all the laws in this section and the next except for the Law of Strict Causality.

2. The conjunction of all the laws in this section and the next except for the Law of Countability.

3. The conjunction of all the laws in this section and the next except for the Law of Finite

Predecession.

Proof. It suffices to consider the five laws stated above, except that for part 3 the Law of Discreteness (or its equivalent) from the next section must be considered because it is a corollary of the law being excluded.

Part 1 is shown by Figure 4. Without the Law of Countability, there may be uncountably many external events, whence part 2. Part 3 is shown by Figure 5, which illustrates an infinite backward chain in the combined ordering having the order type of the negative integers and consisting of alternating arrival and activation ordering links, where each arrival ordering link is taken from a different arrival ordering. ∎

The independence results of Theorem 2 already provide abundant evidence that local laws cannot replace global time in the actor model. To paraphrase the theorem, irreflexivity of the activation and arrival orderings does not imply irreflexivity of the global combined ordering, the local laws do not insure global countability, and finite predecession in the activation and arrival orderings does not imply finite predecession for the combined ordering. Indeed, local discreteness does not imply global discreteness, but that fact will not be stated precisely until the end of the next section and then an entire section will be devoted to its proof.[17]

Independence results similar to Theorem 2 continue to hold even in the presence of ordering laws stronger than those presented in this section. The axiom then becomes merely independent of rather than stronger than the conjunctions of ordering laws, of course. In particular, modulo the replacement of "stronger than" by "independent of", parts 1 and 3 of Theorem 2 remain true in the presence of additional ordering laws forbidding more than one external event and forbidding events that activate infinitely many events.

On the other hand, in the presence of the Law of Discreteness from the next section, the existence of an initial event preceding all other events in the combined ordering implies the Law of Finite Predecession. Thus adding a law postulating such an initial event would require modifying assertion

[17]In §2.4.10 of "Actor systems for real-time computation", Henry Baker gave an example showing that discreteness of two trees does not imply discreteness of the transitive closure of their union. The counterexample to be presented in §7 of this chapter improves upon his result by taking into account the special nature of the activation and arrival orderings.

3 of Theorem 2 so as to exclude the Law of Discreteness and its equivalent, the Law of Finite Chains Between Events in the Combined Ordering, as well as the Law of Finite Predecession.

An independence result that strengthens part 3 of Theorem 2 by allowing locality laws and these additional ordering laws is presented formally in §7.

II.6. The Weak Axiom of Realizability

Now suppose that the strong Axiom of Realizability is replaced by the Weak Axiom of Realizability, so computations are allowed to be infinite in past time as well as in future time. This may seem a strange possibility to consider. Its practical motivation is the fact that some programs are *pure* in the sense that they never change, and properties of such programs may be proved using only the weak axiom.[18] Properties whose proof requires the strong axiom depend upon what has happened in the past, and are usually proved by induction from some initial state. Hence there is a real and useful distinction between properties that require only the weak axiom and those that require the full power of the strong axiom.

The Law of Strict Causality and the Law of Countability remain true under the weak axiom, but the Law of Finite Predecession is replaced by the

Law of Discreteness (LD).[19] *For all events* e_1 *and* e_2, *the set*

$$\{e \mid e_1 \rightarrow e \rightarrow e_2\}$$

is finite.

This law is equivalent to the

Law of Finite Chains Between Events in the Combined Ordering. *There are no infinite chains of events between two events in the combined ordering* $\rightarrow$.[20]

[18] A simple example of such a proof is found in §8.

[19] This was called the Law of Finitely Many Events between two events in the Combined Ordering in a revised version of Carl Hewitt and Henry Baker, "Actors and continuous functionals", MIT LCS Technical Report 194, December 1977. It appeared first in Hewitt and Baker, "Laws for communicating parallel processes", August 1977, but in that paper it was equivalent to the Law of Finite Predecession due to their assumption of an initial event.

[20] A chain is just a linearly ordered set.

Theorem 1. *Assume the Law of Strict Causality. Then the Law of Discreteness is equivalent to the Law of Finite Chains Between Events in the Combined Ordering.*[21]

Proof. The only if direction is trivial.

To prove the converse, assume there are no infinite chains between events in the combined ordering. Then by the totality of arrival orderings, an event has either no predecessors in the arrival ordering of its target, or it has a unique immediate predecessor. Similarly, an event is either external or has a unique immediate predecessor in the activation ordering, namely its activator. Therefore no event has more than two immediate predecessors in the combined ordering.

Now suppose that for some E_1 and E_2 the set $\{ e \mid E_1 \rightarrow e \rightarrow E_2 \}$ is infinite. We will inductively construct an infinite chain, contrary to hypothesis. Let $e_0 = E_2$.

We have a sequence $e_0, \ldots, e_n$ such that

$$E_1 \rightarrow e_n \rightarrow e_{n-1} \rightarrow \cdots \rightarrow e_0 = E_2$$

and $\{ e \mid E_1 \rightarrow e \rightarrow e_n \}$ is infinite. If e_n is not an external event, let E be its activator, and if e_n is not the first event in the arrival ordering for $T(e_n)$ let E' be the unique immediate predecessor of e_n in that arrival ordering. If e_n is not external and $\{ e \mid E_1 \rightarrow e \rightarrow E \}$ is infinite, then define $e_{n+1} = E$. Otherwise E' exists and $\{ e \mid E_1 \rightarrow e \rightarrow E' \}$ is infinite, so define $e_{n+1} = E'$. ∎

This proof is essentially the proof of Konig's Lemma for ordered trees, and does not assume an axiom of choice.[22] Thus the two laws may be interchanged freely. Usually the Law of Finite Chains in the Combined Ordering will be easier to prove, and the Law of Discreteness will seem stronger in use.

The Law of Discreteness also implies the existence of global time functions.[23]

[21]This is a sharpened statement of a fact observed by Hewitt and Baker in the revised version of "Actors and continuous functionals". Since in their paper events could only activate finitely many events, Konig's Lemma could be used in either direction. No proof appears in that paper, but the proof given by Baker in "Actor systems for real-time computation", MIT LCS Technical Report 197, March 1978, fails without the assumption of finite activation. Incidentally, the footnote in "Laws for communicating parallel processes" that says that discreteness is the stronger condition must refer to general orderings.

[22]Raymond Smullyan, *First Order Logic*, Springer-Verlag, New York, 1968. Baker's proof used Konig's Lemma for unordered trees and thus assumed an axiom of choice.

[23]This was observed by Hewitt and Baker in "Laws for communicating parallel processes", but their statement assumes also the existence of an initial event, so for them the Law of Discreteness was equivalent to the Law of Finite Predecession. They also assumed no event could activate infinitely many events.

Theorem 2. *The Weak Axiom of Realizability is equivalent to the conjunction of the Law of Strict Causality, the Law of Countability, and the Law of Discreteness.*

Proof. The weak axiom clearly implies LSC, LC, and LD.

Let $e_0, e_1, e_2, \ldots$ be the events. Define a global time g inductively as follows.

Define $g(e_0) = 0$.

The induction hypothesis for n, $IH(n)$, is the following: $g(e_0), \ldots, g(e_{n-1})$ have been defined so that

1. g is one-to-one.
2. g is integer valued.
3. the combined ordering is preserved.
4. g is already defined on all e_k lying between any two of $e_0, \ldots, e_{n-1}$ in the combined ordering. That is,

$$\forall i, j, k \quad 0 \leq i, j \leq n-1 \ \wedge \ e_i \rightarrow e_k \rightarrow e_j \quad \Rightarrow \quad 0 \leq k \leq n-1.$$

Clearly the fourth part of the induction hypothesis will be impossible to arrange without periodically re-ordering the e_i's, and we must be careful in that re-ordering not to upset the main induction.

Assume $IH(n)$. There are two cases, depending on whether or not e_n is related by $\rightarrow$ to any of $e_0, \ldots, e_{n-1}$. In the simple case, when e_n is not related, define

$$g(e_n) = 1 + \max\{\, g(e_i) \mid 0 \leq i \leq n-1 \,\}.$$

Clearly 1, 2, and 3 of $IH(n)$ hold. Also 4 holds because $\rightarrow$ is transitive and e_n is unrelated to $e_0, \ldots, e_{n-1}$.

Now the hard case, where e_n is related to at least one of $e_0, \ldots, e_{n-1}$. By 4 of $IH(n)$, either e_n precedes all those it is related to, or it follows all those it is related to. Let us say e_n follows all of $e_0, \ldots, e_{n-1}$ that it is related to, since the other possibility is handled in exactly the same fashion. (That is, with arrows reversed, $1 + \max\{\, g(e_i) \mid 0 \leq i \leq n-1 \,\}$ replaced by $\min\{\, g(e_i) \mid 0 \leq i \leq n-1 \,\} - 1$, et cetera.)

If there does not exist a e_k such that $k > n$ and, for some $i, 0 \leq i \leq n-1$, $e_i \rightarrow e_k \rightarrow e_n$ is true, then define $g(e_n) = 1 + \max\{ g(e_i) \mid 0 \leq i \leq n-1 \}$. $IH(n+1)$ then clearly holds. Otherwise we must re-order $\{ e_i \mid i \geq n \}$.

Let

$$\{ e_{k_1}, \ldots, e_{k_m} \} = \bigcup_{i=1}^{n-1} \{ e_k \mid k > n \text{ and } e_i \rightarrow e_k \rightarrow e_n \}.$$

The finiteness of this set is guaranteed by LD. We may assume $k_1 < k_2 < \cdots < k_m$. We re-order the set $\{ e_i \mid n \leq i \leq k_m \}$ by pulling $e_{k_1}, \ldots, e_{k_m}, e_n$ out of it and placing them in front, so that the new order looks like

$$e_{k_1}, e_{k_2}, \ldots, e_{k_m}, e_n, e_{n+1}, \ldots, e_{k_1-1}, e_{k_1+1}, \ldots, e_{k_m-1}$$

and relabel as

$$e'_n, e'_{n+1}, \ldots, e'_{n-1+m}, e'_{n+m}, e'_{n+m+1}, \ldots, e'_{k_1-1+m}, e'_{k_1+1+(m-1)}, \ldots, e'_{k_m}.$$

What has been accomplished by this re-ordering? First of all, nothing has been ruined by it. g is still defined in the same way on the same events, and $IH(n)$ still holds. Some points are now farther back—at most m events farther back—in the new ordering, but if g were to be defined on $e_{k_1}, \ldots, e_{k_m}$ and e_n (newly relabelled $e'_n, \ldots, e'_{n+m-1}, e'_{n+m}$) without any further relabelling of the $e'_i, i \geq n+m$, then every event e'_i would be at least one event closer to being defined than in the original labelling. And in fact it is possible to define g on $e'_n, \ldots, e'_{n+m-1}, e'_{n+m}$ while maintaining the induction hypothesis and without disturbing $e'_i, i \geq n+m$.

Proof of claim: $IH(n)$ still holds, so try again to define g on the n^{th} event, but this time use the new ordering, *ie* define $g(e'_n)$. Relabelling may again be necessary, but no e'_i with $i \geq n+m$ will be relabelled. That is because $e_j \rightarrow e'_i \rightarrow e'_n$ for some $j, 0 \leq j \leq n-1$ would imply $e_j \rightarrow e'_i \rightarrow e_n$ (since $e'_n \rightarrow e_n$), contradicting $e'_i \not\in \{ e_{k_1}, \ldots, e_{k_m} \}$. In fact, several relabellings may be necessary before g becomes defined on an n^{th} event, but these relabellings can only affect the order of $e'_n, \ldots, e'_{n+m-1}$. Each relabelling changes the labels on a smaller initial segment of $\{ e'_i \mid i \geq n \}$, and so finally e'_n becomes such that no $e'_i, i > n$ lies between it and any of $e_0, \ldots, e_{n-1}$ in the

combined ordering. At that point g becomes defined on its n^{th} event. Furthermore g will be defined on all of $e'_n, \ldots, e'_{n+m-1}, e'_{n+m}$ before it is necessary to disturb the labelling above $n+m$, by the same *reductio ad absurdum* as above. Thus the claim.

For each event e_i, therefore, $g(e_i)$ is eventually defined. g is a one-to-one integer valued function that preserves $\rightarrow$, since any non-preservation would show up at a finite stage contrary to the induction. Hence the Weak Axiom of Realizability is satisfied. ∎

The Law of Discreteness has two immediate consequences for the primitive orderings.

Law of Discreteness in the Activation Ordering. *If C is a chain of events in the activation ordering from e_1 to e_2, then C is finite.*

Law of Discreteness in an Arrival Ordering. For all events e_1 and e_2 such that $T(e_1) = T(e_2) = a$, $\{ e \mid e_1 \text{ —}arr_a\rightarrow e \text{ —}arr_a\rightarrow e_2 \}$ is finite.

The first two parts of the following independence theorem are essentially the same as Theorem 2 of §5.

Theorem 3. *The Weak Axiom of Realizability is stronger than the conjunction of*

1. *All the laws in this and the previous section except for the Law of Strict Causality.*
2. *All the laws in this and the previous section except for the Law of Countability.*
3. *All the laws in this and the previous section except for the Law of Discreteness, the Law of Finite Chains Between Events in the Combined Ordering, and the Law of Finite Predecession.*

The third part of this theorem is less obvious, and its proof will be deferred to the next section. It amounts to asserting that the Law of Finite Chains Between Events in the Combined Ordering is independent of the corresponding laws on the activation and arrival orderings. In other words, local discreteness does not imply global discreteness. Hewitt and Baker conjectured that adding additional local laws, which they called *locality laws*, sufficed to derive the Law of Finite Chains Between Events in the Combined Ordering from the corresponding local laws.[24] The next section is devoted to a counterexample.

[24] "Actors and Continuous Functionals".

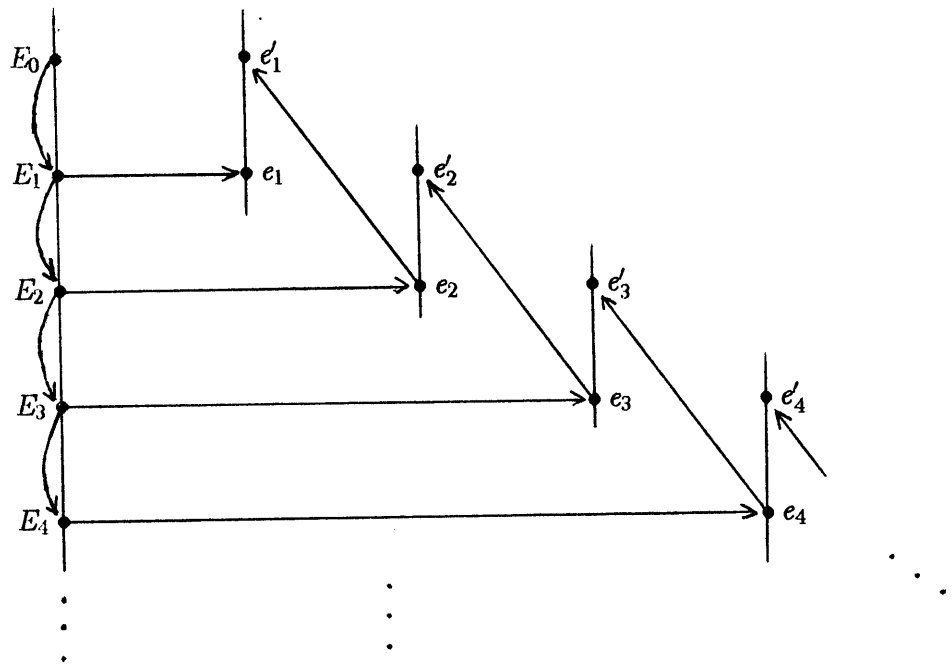


Figure 6. A counterexample to a conjecture by Hewitt and Baker.

II.7. A Strong Independence Result

Figure 6 shows that finite predecession in the activation and arrival orderings does not imply discreteness in the combined ordering. Between any two events in the figure there exists a directed finite path in the combined ordering. In particular, $E_i \rightarrow e_1$ for all i, so there are infinitely many events between E_0 and e_1. In fact, all the events of the figure fall into the infinite chain

$$E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow E_3 \rightarrow E_4 \rightarrow \cdots \rightarrow e'_4 \rightarrow e_4 \rightarrow e'_3 \rightarrow e_3 \rightarrow e'_2 \rightarrow e_2 \rightarrow e'_1 \rightarrow e_1.$$

This proves part 3 of Theorem 3 of the last section.

Consider the finite "top sections" obtained by restricting the diagram in Figure 6 to the events

$$\{E_i \mid i \leq n\} \bigcup \{e_i \mid i \leq n\} \bigcup \{e'_i \mid i < n\}$$

for integers n. While the figure as a whole fails to satisfy the Weak Axiom of Realizability, each top section satisfies the strong Axiom of Realizability and is thus a valid actor event diagram. Not only

are the top sections formally acceptable, but they are physically possible as well. Even supposing that the message of e_1 is sent before the message of e_2 (which is not implied by the fact that their activators occur in that order), it is entirely possible for event e_2 to occur before e_1. That is because messages being sent over computer networks are subject to variable delays from varying route choices and processor loads. While the larger top sections are not very probable, they are still possible. The entire figure is quite impossible, however, instead of being possible with probability zero as extrapolation would suggest.

Figure 6 is the basis for a counterexample to the conjecture that discreteness follows from discreteness in the activation and arrival orderings together with the locality laws discussed in Chapter IV.[25] All that needs to be shown is that acquaintances and creation events can be assigned so that the locality laws are fulfilled. Logically that should await the definition of the locality laws in terms of the structure $\langle \mathrm{E}, \mathrm{A}, T, \text{—}act\text{→}, \mathrm{Arr} \rangle$ and new objects acq, A_0, and $creation$. Illogically it appears here as the proof of a theorem asserting independence of the ordering laws from the locality laws.

Theorem 1. *There exists a structure*

$$\langle \mathrm{E}, \mathrm{A}, T, \text{—}act\text{→}, \mathrm{Arr}, acq, \mathrm{A}_0, creation \rangle$$

of which the Law of Finite Chains in the Combined Ordering is not true, but for which all of the following hold.

1. E *is the set of events.*

2. A *is the set of actors.*

3. T *is the target function:* $\mathrm{E} \rightarrow \mathrm{A}$.

4. $\text{—}act\text{→}$ *is the activation ordering, an irreflexive partial order on* E *such that no event has more than one immediate predecessor.*

5. Arr *is the set of arrival orderings, a set of irreflexive linear orders* $\text{—}arr_a\text{→}$ *on* $T^{-1}(a)$, *for* $a \in \mathrm{A}$.

6. acq *is the acquaintance function:* $\mathrm{E} \rightarrow$ finite-subsets(A).

7. A_0 *is the set of primeval actors.*

[25] Hewitt and Baker, "Actors and continuous functionals".

8. creation is the creation function:$(A - A_0) \rightarrow E$.

9. The Law of Strict Causality.

10. The Law of Countability.

11. The Law of Finite Predecession in the Activation Ordering.

12. The Law of Finite Predecession in an Arrival Ordering.

13. All the locality laws in Chapter IV.

14. There is only one primeval actor. That is, A_0 *is a singleton.*

15. No event is the creation event for infinitely many actors. That is, $\forall e \in E \{a \in A \mid creation(a) = e\}$ *is finite.*[26]

16. No actor ever has more than two acquaintances. That is, $\forall e \in E$ $acq(e)$ *contains at most two actors.*[27]

17. There is an initial event preceding all other events in the activation ordering.[28]

18. No event activates infinitely many events.[29]

Proof. The events are, as in Figure 6,

$$E = \{E_i \mid i \geq 0\} \bigcup \{e_i \mid i \geq 1\} \bigcup \{e'_i \mid i \geq 1\}.$$

Of course this is just a set of names. Let

$$A = \{a_i \mid i \geq 0\}$$

be the set of actors, also a set of names. The target function is defined by

$$T(E_i) = a_0, \ i \geq 0;$$
$$T(e_i) = a_i, \ i \geq 1;$$
$$T(e'_i) = a_i, \ i \geq 1.$$

[26]This was a law in "Actors and continuous functionals".

[27]Baker, "Actor systems for real-time computation", required that the number of acquaintances of an actor be bounded.

[28]This was postulated for simplicity in Hewitt and Baker, "Laws for communicating parallel processes".

[29]This was a law in "Actors and continuous functionals".

The activation ordering is defined by

$$\begin{aligned} E_i &\text{ —}act\text{→ } E_j,\ 0 \leq i < j; \\ E_i &\text{ —}act\text{→ } e_j,\ 0 \leq i \leq j, j \geq 1; \\ E_i &\text{ —}act\text{→ } e'_j,\ 0 \leq i \leq j+1, j \geq 1; \\ e_{i+1} &\text{ —}act\text{→ } e'_i,\ i \geq 1. \end{aligned}$$

The arrival orderings $\text{—}arr_{a_i}\text{→}$, $a_i \in$ A, are defined by

$$\begin{aligned} E_i &\text{ —}arr_{a_0}\text{→ } E_j,\ 0 \leq i < j; \\ e'_i &\text{ —}arr_{a_i}\text{→ } e_i,\ i \geq 1. \end{aligned}$$

The acquaintance function is defined by

$$\begin{aligned} acq(E_0) &= \{ a_0 \}; \\ acq(E_i) &= \{ a_0, a_i \},\ i \geq 1; \\ acq(e_i) &= \emptyset,\ i \geq 1; \\ acq(e'_i) &= \emptyset,\ i \geq 1. \end{aligned}$$

The only primeval actor is a_0, so $A_0 = \{ a_0 \}$. The other actors are created in the course of computation, and their creation events are defined by

$$creation(a_i) = E_{i-1},\ i \geq 1.$$

The structure so defined confirms the claims of the theorem. ∎

Describing this pseudo-computation informally, there is only one actor a_0 that exists at the beginning. The initial event E_0 tells it to begin. It then creates a_1 and sends a message to itself. When that message arrives in event E_1, it creates a_2, sends a message to a_2 telling it about a_1, forgets about a_1, and sends another message to itself. When that message arrives in event E_2, it creates a_3, sends a message to a_3 telling it about a_2, forgets about a_2, and sends another message to itself. In general, when a message from itself arrives in an event E_i, actor a_0 creates a_{i+1}, sends a message to a_{i+1} telling a_{i+1} about a_i, forgets about a_i, and sends another message to itself. It does this forever, so the computation cannot terminate.

Each created actor a_i, $i \geq 1$, upon receiving a message naming an actor, sends a message to that actor. The content of the message is irrelevant.

Figure 7 defines these actor behaviors using a toy programming language.

```
(master = acq initially [ ]
          inside
          accept [ ]
          (create ((slave = accept [ x ]
                            if actorp(x)
                               then send "ignore" to x
                               else dummy))
              (if equal [ acq [ ] ]
                  then dummy
                  else send acq to slave) ;
              change acq to slave) ;
          send [ ] to master)
```

Figure 7. A program to illustrate the counterexample.

It is possible for a_0's message to a_1 to be slow, so that event E_1 occurs, a_2 is created and receives the message about a_1, and the message from a_2 arrives at a_1, all before a_0's message arrives at a_1. In that way e_1' can precede e_1 in the arrival ordering of a_1. This scenario can occur at any number of actors, even infinitely many. Figure 7 shows it occurring at all actors, however, and that cannot be.

Figure 7 can be seen to be impossible only when it is considered as a whole. This shows the "globalness" of the phenomenon, and that a truly global law, such as the Law of Discreteness, must be devised to take care of it.

Upon learning of this counterexample, Professor Hewitt set the problem of finding a counterexample as an exercise for MIT subject 6.835. Valdis Berzins solved the exercise, finding a different, symmetric counterexample.[30]

II.8. Modifying a Proof

One of the purposes of this chapter has been to relax unnecessary restrictions on the actor event diagrams. As noted at the end of §3, there is good reason to allow an event to activate infinitely many events. This was not allowed by Hewitt and Baker, partly because they wished to assume finite activation in proofs, and partly for reasons mentioned at the end of §4. Having removed the assumption

[30]Valdis Berzins, "An independence result for actor laws", MIT LCS Computation Structures Group Note 34, December 1977.

of finite activation from Theorem 1 of §6, it is now time to remove that assumption from the main theorem of "Actors and continuous functionals".

Considerable notation and some definitions from that paper will be needed before proving the lemma that depended upon finite activation.

Messages must be represented in some language, and have some kind of structure. For the purposes of this proof there are two sorts of messages, corresponding to two kinds of events. A *request event* is an event of the form

$$[f \leftarrow \texttt{request:}x, \texttt{reply-to:}c]$$

which represents passing an argument x to the actor f, with instruction to send any result to a continuation actor c. A *reply event* is an event of the form

$$[c \leftarrow \texttt{reply:}y]$$

which represents the arrival of a result y at the continuation actor c. By convention, replies are responses to previous request events.

Definition. *If an event* e_1 *is of the form*

$$[\ldots \leftarrow \texttt{request:}\ldots, \texttt{reply-to:}c],$$

e_2 *is of the form*

$$[c \leftarrow \texttt{reply:}\ldots],$$

$e_1 \text{ —}act\text{→ } e_2$, *and for no event* e *of the same form as* e_2 *is* $e_1 \text{ —}act\text{→ } e \text{ —}act\text{→ } e_2$ *true, then* e_2 *is said to be a* reply *to* e_1.

A request event may have no replies, one reply, nineteen replies, or infinitely many replies. For a request event whose target is an actor that behaves as a *procedure*, however, there is at most one reply, by definition.[31]

For an event e let $R(e) = \{e\} \bigcup \{e' \mid e \rightarrow e'\}$ and $L(e) = \{e\} \bigcup \{e' \mid e' \rightarrow e\}$.

[31] See "Actors and continuous functionals".

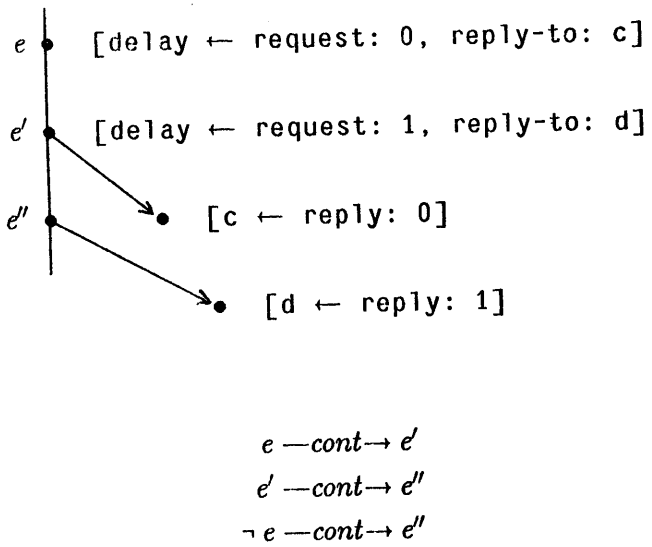


Figure 8. The continuation ordering may not be transitive.

Definition. *If e is a request event then the* activity *corresponding to e is*

$$R(e) \bigcap (\bigcup \{ L(e') \mid e' \text{ is a reply to } e \}).$$

Perhaps not all events in the activity corresponding to e actually contribute to answering the request e, but certainly all events that do contribute are in the activity. An activity may not be finite, because a request can have infinitely many replies. If a request has only finitely many replies, though, as is the case if its target is a procedure, then its activity is guaranteed to be finite by the Law of Discreteness.

Definition. *If e and e' are events, $e \rightarrow e'$, and there is some activity α such that $e, e' \in \alpha$, then we say $e \text{ —}cont\text{→ } e'$.*

Although $\text{—}cont\text{→}$ is called the continuation ordering, it is not in general a true ordering because it may not be transitive. In Figure 8, $e \text{ —}cont\text{→ } e'$ and $e' \text{ —}cont\text{→ } c''$, but $e \text{ —}cont\text{→ } e''$ is not

true. The continuation ordering is transitive when restricted to activities corresponding to requests of a procedure, though, because by definition the activites of a procedure are properly nested. Note that $\text{—}cont\text{→}$ is a subrelation of the combined ordering →.

An actor that is a procedure and initiates the same activity, in the sense of the same messages with the same targets and the same relationships between events, whenever it is sent the same request is said to behave like a *function.*

The definition of an *immediate f-descendant* in the first version of "Actors and continuous functionals"[32] contained a small but subtle error that was partially corrected in subsequent versions.[33] The idea is that the immediate f-descendants of $\langle x, y\rangle \in \text{graph}(f)$ are those $\langle x', y'\rangle \in \text{graph}(f)$ that must be known in order to compute $f(x)$ without recursing. As is often the case, the proof is correct because it depends on what the definition is supposed to be, not its formal specification. The definition below is supposed to be what the definition was supposed to be.

Definition. *Suppose an actor f behaves like a mathematical function, $\langle x, y\rangle \in \text{graph}(f)$, and $\langle x', y'\rangle \in \text{graph}(f)$. Then $\langle x', y'\rangle$ will be said to be an* immediate f-descendant *of $\langle x, y\rangle$ if there is some history of f that has events e_1 and e_2 of the form*

$$e_1\text{:}[f \leftarrow \texttt{request:}x, \texttt{reply-to:}\ldots]$$

$$e_2\text{:}[f \leftarrow \texttt{request:}x', \texttt{reply-to:}\ldots]$$

such that e_2 belongs to the activity initiated by e_1 (so that e_1 $\text{—}cont\text{→}$ e_2) and it is not the case that there is an event e of the form

$$e\text{:}[f \leftarrow \texttt{request:}\ldots, \texttt{reply-to:}\ldots]$$

such that e_1 $\text{—}cont\text{→}$ e $\text{—}cont\text{→}$ e_2.

Definition. *Suppose that $\langle x, y\rangle \in \text{graph}(f)$. Then*

$$\text{immediate-descendants}_f(\langle x, y\rangle) = \{\, \langle x', y'\rangle \mid \langle x', y'\rangle \textit{ is an immediate f-descendant of } \langle x, y\rangle \,\}.$$

[32]IFIP Working Conference on Formal Description of Programming Concepts, August 1977, 16.1–16.21.

[33]MIT LCS Technical Report 194, December 1977.

As an example, Hewitt and Baker give the following procedure.

```
fib(n) ≡
        if n=1 then 1
        if n=2 then 1
        if n>2 then fib(n-1)+fib(n-2).
```

$$\text{immediate-descendants}_{fib}(\langle 1, 1\rangle) = \emptyset$$
$$\text{immediate-descendants}_{fib}(\langle 2, 1\rangle) = \emptyset$$
$$\text{immediate-descendants}_{fib}(\langle 3, 2\rangle) = \{\langle 1, 1\rangle, \langle 2, 1\rangle\}$$
$$\text{immediate-descendants}_{fib}(\langle 5, 5\rangle) = \{\langle 3, 2\rangle, \langle 4, 3\rangle\}$$

Now the only real use Hewitt and Baker make of the assumption of finite activation is in proving the following lemma.

Lemma 1. *If an actor f behaves like a mathematical function and $\langle x, y\rangle \in \text{graph}(f)$, then* $\text{immediate-descendants}_f(\langle x, y\rangle)$ *is finite.*

Proof. Let e_1 be a request for the procedure f to compute the value $f(x)$. That is, e_1 is of the form

$$e_1{:}[f \leftarrow \texttt{request:}x, \texttt{reply-to:}\ldots].$$

By the way Hewitt and Baker define "function" there can be at most one reply to this request. There is a reply, since $\langle x, y\rangle \in \text{graph}(f)$, so call it e_2. Since e_1 has a unique reply, the activity initiated by e_1 is just $\{e_1, e_2\} \bigcup \{e \mid e_1 \rightarrow e \rightarrow e_2\}$. This set is finite by the Law of Discreteness, and so $\text{immediate-descendants}_f(\langle x, y\rangle)$ is finite by the definition. ∎

The lemma thus remains true without the assumption of finite activation. As this lemma is the only place in its proof where Hewitt and Baker use finite activation, the theorem to be stated below no longer depends upon that hypothesis.

Definition. *If G is a set of input-output pairs, then*

$$D_f(G) = \{\langle x, y\rangle \mid \langle x, y\rangle \in \text{graph}(f) \textit{ and } \text{immediate-descendants}_f(\langle x, y\rangle) \subseteq G\}.$$

Theorem 2. (Hewitt and Baker.) *If an actor f behaves like a mathematical function, then D_f is a continuous functional in the sense of Scott, and* graph(f) *is the limit of D_f beginning with the empty graph. Also* graph(f) *is the minimal fixed point of D_f.*

Chapter III

Nondeterminism

Is the universe deterministic? Regardless of the answer, there exist systems so complex that their unique future behavior cannot exactly be predicted in any practical sense. In practice such systems are considered nondeterministic.

This chapter deals with the semantics of nondeterministic programming languages. The usual way of representing nondeterminism in a denotational fixed point semantics is by means of power domains, so called by analogy with power sets. Extending the power domain construction to apply to incomplete domains makes possible a power domain semantics for nondeterministic programming languages in which a fair merge can be written.

III.1. Nondeterminism can be Viewed as Incomplete Specification

Abstraction is essential to understanding complex systems. One difference between good and bad programmers is that good programmers think in terms of the function performed by a program segment whereas bad programmers are likely to think of the program segment as a sequence of steps. Programming language semantics seeks to provide abstract descriptions of program segments.

As part of the abstraction process, details are suppressed. One detail universally suppressed by programming language semantics is the amount of time required to do a particular thing, since it varies from implementation to implementation or even from moment to moment. As a result programming language semantics cannot always say exactly what the output of a program with concurrency will be, because the output may depend upon timing. Abstraction can therefore lead to nondeterminism.

Nondeterminism can result from any incomplete specification of a programming language, whether deliberate as in the case of abstraction or accidental as in the case of oversight. Though the nondeterministic program given below is written in APL, it uses no special features of that language. Almost any popular programming language would have served. APL was chosen partly because it is simple, concise, and well-known, but the main reason is that an ambiguity in APL's order of evaluation went unrecognized for many years, the ambiguity created significant nondeterminism, and the ambiguity was of the sort that can be exploited through concurrency.

Consider the program FOO defined by

```
      ∇ RESULT ← FOO X
[1]   GLOBAL ← 0
[2]   RESULT ← (F X) + (G X)
      ∇
```

which, given an argument X, sets the global[1] variable GLOBAL to 0 and then returns as its result the sum of F X and G X, where F and G are user defined "function" subprograms. If F and G do any significant computation at all, then the time required to execute FOO on a sequential machine is the sum of the execution times for F and G. For example, if F and G each take one minute to return their answers, then executing FOO takes two minutes. With the advent of multiprocessors capable of performing several independent computations concurrently, it has become feasible to consider evaluating F and G at the same time on separate processors, so that executing FOO might take as little as half the time required when only a single processor is used.

This example suggests one of the speed gains possible through multiprocessing. The particular

[1]In the sense that the memory location denoted by GLOBAL is accessible to subprograms invoked by FOO. The example is indifferent to the question of whether the memory location can be directly accessed by all hardware processors.

speedup illustrated is possible any time that two or more arguments to a function each require significant time to evaluate. Devotees of largely functional languages such as Lisp and APL perceive this to be of profound importance for the design of languages intended for execution on multiprocessors.[2]

Nondeterminism often accompanies this and many other techniques for concurrency. That is, the outcome of a program may no longer be completely determined. Nondeterminism may or may not affect the usefulness or correctness of a program. Consider, for example, a program that conducts a parallel search for a proof of or a counterexample to the Goldbach conjecture. It does not matter which particular proof or counterexample is first found. While some programs must be deterministic to be correct, nondeterminism has a role in artificial intelligence programs and programs such as operating systems that depend on inputs presented at unpredictable times.

Even so simple a program as FOO can be nondeterministic. Suppose the subprograms F and G invoked by FOO are defined as follows.

```
      ∇ RESULT ← F X
[1]   RESULT ← GLOBAL
[2]   GLOBAL ← 1
      ∇

      ∇ RESULT ← G X
[1]   RESULT ← GLOBAL
[2]   GLOBAL ← 1
      ∇
```

Aside from their names, these programs are identical. Each reads the global variable GLOBAL and, after setting GLOBAL to 1, returns the value read as its result.

On a sequential machine, these definitions cause FOO to evaluate to 1 (regardless of the value of X). Here is what happens. First FOO sets GLOBAL to 0. Then, in line 2 of FOO, G is invoked with X as its argument.[3] G reads the global variable GLOBAL, finds its value to be 0, sets GLOBAL to 1, and returns 0 as its result. Then F is invoked with argument X. F reads GLOBAL, finds its value to be 1, sets

[2]See for example Friedman and Wise, "Aspects of applicative programming for parallel processing", *IEEE Transactions on Computers* C-27, 4, April 1978, pages 289–296.

[3]Since APL as now defined evaluates right-most arguments first.

GLOBAL to 1, and returns 1 as its result. FOO then sums the results of F and G to obtain its result: 1. On an interactive APL terminal:

```
    X ← 297      (or other arbitrary value)
    FOO X
1
```

Were F X and G X to be evaluated in parallel on a multiprocessor, FOO X might sometimes return 0 instead of 1. The reason is that F and G might both read the global variable GLOBAL before the second line of either subprogram had been executed to set GLOBAL to 1. Thus both F and G might return 0 as their result. On an APL terminal:

```
    FOO X
0
```

would be possible as well as

```
    FOO X
1
```

A given multiprocessor implementation might consistently return a particular one of these two possible results. Nonetheless the program must be regarded as nondeterministic, since the program itself does not determine a unique answer; only when the program is paired with an implementation can the result be determined. Indeed the result may not be determined even then, since the result may be affected by dynamically changing conditions within the multiprocessor. For example, the number of processors available to a computation can change in response to resource requests by concurrent computations.

Thus parallel evaluation of arguments can lead to nondeterminism because the order in which events occur in global time is left incompletely specified. Were the semantics of a program to determine completely the order of events in global time, the program would be sequential; it is when the semantics least constrains the order of events that there exist the greatest opportunities for parallelism. Opportunities for parallelism therefore arise from a kind of semantic ambiguity regarding the

order of events in global time. Until recently, for example, the (informal) semantics of APL did not prescribe a definite order of evaluation of expressions, so that some APL implementations evaluated the expression

```
X - (X ← X - 1)
```

from left to right and obtained 1, while other equally correct APL implementations evaluated from right to left and obtained 0.[4] Had a multiprocessor implementation of APL existed, it could have evaluated the subexpressions `X` and `(X ← X - 1)` in parallel, obtaining 0 on some occasions and 1 on others. This ambiguity in the semantics of APL has now been fixed by the adoption of a standard order of evaluation, but the remedy precludes the parallel evaluation of arguments that was allowed by the ambiguity of the old semantics.

What if the programmer intends a program to be deterministic? Then the programmer must arrange for the sequence-sensitive portions of the program to be executed in a definite order. `FOO`, for example, could be rewritten as

```
      ∇ RESULT ← FOO2 X; LOCAL1; LOCAL2
[1]   GLOBAL ← 0
[2]   LOCAL1 ← F X
[3]   LOCAL2 ← G X
[4]   RESULT ← LOCAL1 + LOCAL2
      ∇
```

using local variables to hold the results of evaluating `F X` and `G X`. `FOO2` is deterministic even when function arguments are evaluated in parallel. Remember that the possibility of evaluating arguments in parallel was not considered when the APL language was designed, and even so the only reason `F` and `G` cause problems when evaluated in parallel is that each assigns to a global variable referenced by the other. Most well-written subprograms have no such side effects. In a language designed specifically for concurrency, troublesome side effects could be expected to be even rarer.

Not only is some incompleteness in specifying the order of events in global time desirable because it allows concurrency, but it is necessary when concurrency is allowed. For a programming

[4]Richard H Lathwell, "Some implications of APL order-of-execution rules", APL79, *APL Quote Quad* 9, 4-Part 1, June 1979, pages 329–332.

language semantics to specify completely the order in which events are to occur during multiprocessor execution of a concurrent program is generally impossible, since it would entail fixing myriad details such as the number and relative speeds of concurrent processors, the exact times of and delays occasioned by page faults and other interrupts, the timing of signals between processors and the manner in which they are arbitrated, and so on, down to the levels of time resolution at which quantum indeterminacy becomes important.

Except for two general requirements, the actor model specifies none of these timing details. The first requirement is that in keeping with the idea of actors as independent computational agents each actor has the computing energy[5] it needs to process messages sent to it. The second requirement is that every message eventually arrives at its target, a requirement known as *finite delay*. These requirements leave much unsaid about the order of events in an actor's arrival ordering. The nondeterminism that results will be called *arrival nondeterminism.*

Arrival nondeterminism is similar to the notion of global nondeterminism introduced by Francez *et al* for the programming language CSP,[6] but §8 points out an important difference. The local nondeterminism of CSP is a form of the choice nondeterminism discussed below.

Choice nondeterminism arises from the presence of choice points within a program, where an implementation is allowed to choose the program's flow of control at random from among a finite set of alternatives. The implementation does not have to make the choice randomly, but it may. Dijkstra's guarded commands are examples of such choice points. Although choice points permit concurrency, they have the defect of permitting random choice as well. Choice points are of interest in this dissertation only because they are often used to model the nondeterminism that accompanies concurrency. Nondeterministic concurrency differs from random choice, but using choice points to model nondeterministic concurrency reduces the problem of providing a semantics for nondeterministic concurrent programs to the problem of generalizing the existing theory of semantics for sequential programs to handle choice points. It is then important to remember that in this context choice points

[5]Computing energy is computing power integrated over time. If several actors share time on a single processor, for example, an actor's computing energy is the computing power of the processor multiplied by the time that the actor actually uses the processor.

[6]Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, 1979, pages 190-308.

are only an attempt to model concurrency. If theories of sequential programs with choice points turn out to produce different conclusions about concurrent programs than theories based directly upon true concurrency, then the idea that concurrency can be modelled by choice points must yield. That has turned out to be the case. Considering concurrency directly leads to regarding some programs as *unboundedly nondeterministic*, but it can be shown that no sequential program with random choice points is unboundedly nondeterministic.

Unbounded (infinite) nondeterminism is a property of programs that on some fixed input are certain to return an answer, but the set of possible answers is infinite. Unbounded nondeterminism will be considered at some length in §7 and §8, but its present importance is that a plausible theory of semantics for concurrent computation must differ from a theory of semantics for sequential programs with choice points.

The next three sections present the mathematical foundation underlying a theory of semantics for concurrent computation.

III.2. Fixed Point Semantics

The denotational theory of programming language semantics is concerned with finding mathematical objects that represent what a program does. Examples of such objects are partial functions, sequences of states, and actor event diagrams. Usually there is a partial ordering $\leq$ on these objects with $x \leq y$ meaning that x is compatible with but possibly less defined than y. In other words, x approximates y. If the objects are partial functions, for example, $f \leq g$ may mean that f agrees with g on all values for which f is defined. If the objects are actor event diagrams, $x \leq y$ means x is a possible initial history[7] of y. The object representing a program P is found by solving an equation of the form $x = f_P(x)$. This section states conditions guaranteeing a solution to that equation.

Let $\langle D, \leq \rangle$ be a partially ordered set, and let A be a subset of D. $a \in A$ is a *minimal* element of A iff A contains no elements below a. That is,

$$\forall x \in A \quad x \leq a \Rightarrow x = a.$$

[7]See §IV.3.

$a \in A$ is a *least* element of A iff a lies below every other element of A. That is,

$$\forall x \in A \quad a \leq x.$$

Maximal and *greatest* elements are defined dually.

An *upper bound* for $A \subseteq D$ is an element $u \in D$ such that

$$\forall x \in A \quad x \leq u.$$

A *least upper bound* for $A \subseteq D$ is an upper bound that is least in the set of upper bounds for A in D. (Least upper bounds are sometimes called *limits*, because they are a special case of colimits in category theory; there is also a T_0 topology on D in which the least upper bound of an increasing sequence is a limit of the sequence in the topological sense.) In general, a set may not have upper bounds, and a set may have upper bounds but no least upper bound. Examples are the rationals $\mathbf{Q}$ under the usual ordering, and the negative rationals as a subset of $D = \mathbf{Q} - \{0\}$. If a set has a least upper bound, though, it has exactly one. The least upper bound of $A \subseteq D$ will be written $\bigvee_{x \in A} x$ or $\bigvee A$, except that $\sqsubseteq$ and $\bigsqcup$ will sometimes be used in place of $\leq$ and $\bigvee$.

A set $A \subseteq D$ is *directed* iff every pair of elements of A has an upper bound in A. It then follows that every finite subset of A has an upper bound in A. These upper bounds need not be least. For example, suppose D is the power set of the natural numbers ω ordered by inclusion. Then the set of all finite subsets of ω is directed, as is the three element set consisting of $\{0\}$, $\{1\}$, and ω.

Let $\langle D, \leq \rangle$ and $\langle D', \leq' \rangle$ be partially ordered sets. A function $f{:}D \to D'$ is *monotonic* iff it preserves order, so that $\forall x, y \in D$

$$x \leq y \Rightarrow f(x) \leq' f(y).$$

f is *ω-continuous* iff it is monotonic and preserves all existing least upper bounds of countable increasing sequences, so that if $\{x_i\}_{i \in \omega}$ is a sequence in D with $x_i \leq x_{i+1}$ for all $i \in \omega$ then

$$f(\bigvee_{i \in \omega} x_i) = \bigvee'_{i \in \omega} f(x_i).$$

(Equivalently, f is ω-continuous if it preserves least upper bounds of countable directed sets.) Note that this definition does not presume that all countable increasing sequences have least upper bounds, but states only that f preserves those least upper bounds that exist.

$\langle D, \leq \rangle$ is *ω-complete* iff every countable increasing sequence (equivalently every countable directed set) has a least upper bound in D. That is, if for all $i \in \omega$ $x_i \in D$ and $x_i \leq x_{i+1}$, then $\bigvee_{i \in \omega} x_i$ exists.

Now suppose $\langle D, \leq \rangle$ has a least element $\perp$ and is ω-complete. Then every ω-continuous function $f{:}D \rightarrow D$ has a fixed point given by $\bigvee_{i \in \omega} f^i(\perp)$, and furthermore this fixed point is least among all fixed points of f.

This is the most basic fact of fixed point semantics. Typically D is a set of possible meanings for programs, such as a set of partial functions from inputs to outputs, ordered according to some approximation ordering. The semantics of a programming language defines for each program P a continuous function $f_P{:}D \rightarrow D$. The program P is then said to *denote* the least fixed point of its associated continuous function f_P. The domain D must be ω-complete to ensure that the least fixed point exists.

For more information on fixed point semantics, readers should consult the tutorial article by Tennent,[8] the textbook by Stoy,[9] or the comprehensive volumes of Milne and Strachey.[10] These references deal only with fixed point semantics on lattices, however, while we must consider more general partial orders.

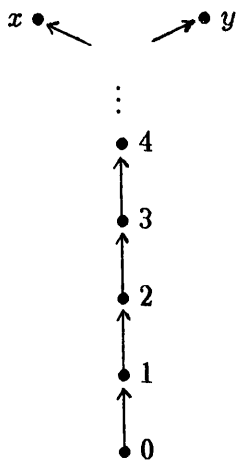
III.3. Domains and Their Completions

Usually there is an intuitive sense in which some elements of the partially ordered sets considered by fixed point semantics are finite. They may be partial functions defined for only finitely

[8]R D Tennent, "The denotational semantics of programming languages", *CACM* **19**, 8, August 1976, pages 437–453.

[9]Joseph E Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Cambridge MA, 1977.

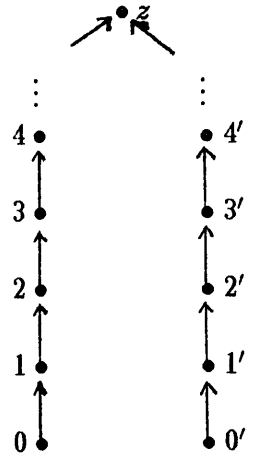
[10]Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London, 1976.



$$D = \{0, 1, 2, 3, \ldots\} \bigcup \{x, y\}$$

$$i \sqsubseteq j \quad \text{if } i \leq j$$
$$i \sqsubseteq x$$
$$i \sqsubseteq y$$

Figure 1. A partial order in which every element is isolated.



$$D = \{0, 1, 2, 3, \ldots\} \bigcup \{0', 1', 2', 3', \ldots\} \bigcup \{z\}$$

$$i \sqsubseteq j \quad \text{if } i \leq j$$
$$i' \sqsubseteq j' \quad \text{if } i \leq j$$
$$i \sqsubseteq z$$
$$i' \sqsubseteq z$$

Figure 2. A partial order in which no element is isolated.

many values, for example, or they may be finite partial computations. This sense of finiteness lies behind the following abstract definition.

Let $\langle D, \leq \rangle$ be a partially ordered set. An element $x \in D$ is *isolated* iff whenever $A \subseteq D$ is directed, $\bigvee A$ exists, and $x \leq \bigvee A$, there exists $a \in A$ with $x \leq a$. In other words, x is isolated if one must go through x in order to get up to or above x via the limit process. As examples, the finite sets

are the isolated elements of the power set of ω ordered by inclusion; the ordinal $\omega + 1$ is isolated in the set of countable ordinals under the usual ordering; in the partial order of Figure 1, every element is isolated, while in the partial order of Figure 2 no elements are isolated.

The least element of a partially ordered set is always isolated provided it exists. 0 is the least element of the nonnegative rationals under the usual ordering, and it is also the only isolated element. The entire set of rationals has no isolated elements under the usual ordering.

For purposes of programming language semantics, partially ordered sets with least elements form too general a category. The partially ordered sets of greatest interest for computer science are those whose isolated elements are *dense* in the sense that every element is a least upper bound of a countable set of isolated elements. To avoid transfinite inductions, and to make directed completeness equivalent to ω-completeness, it is convenient to assume also that there are only countably many isolated elements.

Definition. *A* domain *is a partially ordered set* $\langle D, \leq \rangle$ *such that*

1. D *has a least element* $\perp$.

2. Every element of D *is the least upper bound of a countable increasing sequence of isolated elements.*

3. The isolated elements of D *are countable.*

This definition is nonstandard. The standard definition requires also that D be ω-complete, so that ω-continuous functions from D to D will have fixed points.

An ω-complete domain is *complete* in the sense that every directed subset has a least upper bound. An ω-complete domain is also known as a countably algebraic complete partial order.[11]

Every domain D can be embedded in an ω-complete domain $\overline{D}$ that is, in a precise sense, the smallest ω-complete domain containing D. The isolated elements of $\overline{D}$ are precisely the isolated elements of D,[12] but in general $\overline{D}$ contains limit points that are not found in D. $\overline{D}$ is uniquely

[11] M B Smyth, "Power domains", *J Computer and System Sciences* **16**, 1978, pages 23–36.

[12] Hence $\overline{D}$ differs from completions that do not preserve least upper bounds, such as the basis completion (Markowsky and Rosen) and Bloom's ω-completion. $\overline{D}$ is isomorphic to the basis completion of $\{ x \in D \mid x \text{ is isolated in } D \}$. See G Markowsky and B K Rosen, "Bases for chain-complete posets", *IBM J Research and Development* **20**, 2, March 1976, pages 138–147, Stephen L Bloom, "Varieties of ordered algebras", *J Computer and System Sciences* **13**, 2, October 1976, pages 200–212, and Daniel Lehmann, "On the algebra of order", *J Computer and System Sciences* **21**, 1, August 1980, pages 1–23.

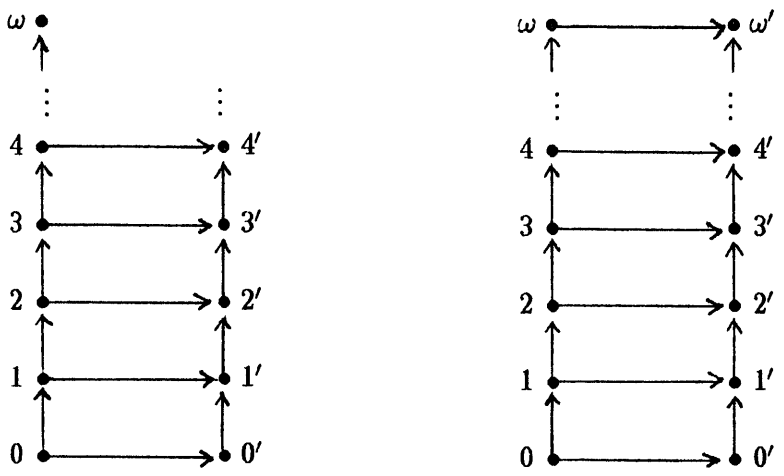


Figure 3. A domain and its ω-completion.

determined up to isomorphism, and is called the *ω-completion*, or simply the *completion*, of D. It will be shown that for any domain D the power domain of D is isomorphic to the power domain of its completion $\overline{D}$. Then why not use ω-complete domains only, as is standard? Because the power domain is interpreted with reference to the domain from which it is built. As will be explained in §5, the underlying domain is incomplete in actor semantics.

At this point readers may wish to read the definition of the closure operation ^{c} on the next page and then skip to §4. The remainder of this section shows how $\overline{D}$ may be constructed, and proves the facts mentioned above.

As an aid to understanding the concrete construction of $\overline{D}$ that follows, consider the domain

$$\langle \{0, 1, 2, 3, \ldots\} \bigcup \{\omega\} \bigcup \{0', 1', 2', 3', \ldots\}, \sqsubseteq \rangle$$

where

$$\begin{array}{ll} i \sqsubseteq j & \text{if } i \leq j \\ i' \sqsubseteq j' & \text{if } i \leq j \\ i \sqsubseteq j' & \text{if } i \leq j \\ i \sqsubseteq \omega & \end{array}$$

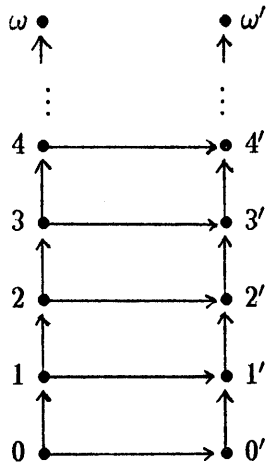


Figure 4. An incomplete domain.

This domain is pictured in Figure 3, along with its intuitive completion. Figure 4 shows why ω must be less than ω'. The domain in Figure 4 is incomplete because the increasing sequence $\{ i \}_{i \in \omega}$ has ω and ω' as its upper bounds, but neither is least.

Let $\langle D, \leq \rangle$ be a domain.

Definition. *The* closure *of* $A \subseteq D$ *is*

$$A^c = \{ d \in D \mid \exists X \subseteq D,\ X \text{ directed},\ d = \bigvee X,\ \text{and} \forall x \in X\ \exists a \in A\ x \leq a \}.$$

Lemma 1. *If* $a, b \in D$ *are isolated and have an upper bound* d*, then they have an isolated upper bound* c *such that* $c \leq d$.

Proof. Let $\{ d_i \}_{i \in \omega}$ be an increasing sequence of isolated elements with $\bigvee_{i \in \omega} d_i = d$. $\{ d_i \mid i \in \omega \}$ is directed, so there exist d_i and d_j with $a \leq d_i$ and $b \leq d_j$. Let $k = \max\{ i, j \}$ and $c = d_k$. ∎

Lemma 2. *If* $Y \subseteq D$ *is directed, and* $x = \bigvee Y$*, then there exists a directed set* Z*, consisting solely of isolated elements, such that*

$$x = \bigvee Z$$

and

$$\forall z \in Z\ \exists y \in Y\ z \leq y.$$

Proof. For $y \in Y$ let Z_y be a directed set of isolated elements with $y = \bigvee Z_y$, and let

$$Z = \bigcup_{y \in Y} Z_y.$$

It is clear that Z consists of isolated elements, and that $\forall z \in Z\ \exists y \in Y\ z \leq y$.

Let $z_1, z_2 \in Z$, and let $y_1, y_2 \in Y$ be such that $z_1 \in Z_{y_1}$ and $z_2 \in Z_{y_2}$. Let $y_3 \in Y$ be an upper bound for y_1 and y_2, and hence for z_1 and z_2. By Lemma 1 there exists an isolated $z \in D$ such that $z_1, z_2 \leq z \leq y_3 = \bigvee Z_{y_3}$. Let $z_3 \in Z_{y_3}$ be such that $z \leq z_3$. z_3 is an upper bound for z_1 and z_2 in Z, so Z is directed.

Clearly x is an upper bound for Z.

Let x' be an upper bound for Z. x' is an upper bound for each Z_y, so $y = \bigvee Z_y \leq x'$. Thus x' is an upper bound for Y, whence $x = \bigvee Y \leq x'$. Therefore x is the least upper bound of Z. ∎

Lemma 3. *The map* ^{c} *is a closure operator on the power set of* D.

Proof. If $A \subseteq B$, then $A^c \subseteq B^c$. Also $A \subseteq A^c$.

To show $(A^c)^c = A^c$, let $x \in (A^c)^c$. A^c is downward-closed. That is, if $a \in A^c$ and $x \leq a$ then $x \in A^c$. Therefore there exists a directed set $Y \subseteq A^c$ with $x = \bigvee Y$. Let Z be a directed set of isolated elements with $x = \bigvee Z$ and $Z \subseteq A^c$.

Let $z \in Z$. Since $z \in A^c$ there exists a directed set W such that $z = \bigvee W$ and $\forall w \in W\ \exists a \in A\ w \leq a$. Since z is isolated there is some $w \in W$ with $z \leq w$. Hence there exists $a \in A$ such that $z \leq a$.

Z is directed, $x = \bigvee Z$, and $\forall z \in Z\ \exists a \in A\ z \leq a$. Consequently $x \in A^c$. ∎

Note that A^c is downward-closed and is closed under existing least upper bounds in D of directed subsets.

Definition. *Let* $\langle D, \leq \rangle$ *be a domain with least element* $\perp$. *Its* completion *is* $\langle \overline{D}, \sqsubseteq \rangle$, *where*

$$\overline{D} = \{ A^c \mid \perp \in A \subseteq D,\ A \text{ directed} \}$$

and for all $A, B \in \overline{D}$

$$A \sqsubseteq B \Leftrightarrow A \subseteq B.$$

This makes $\overline{D}$ a partial order. Generally $\langle \overline{D}, \sqsubseteq \rangle$ is not a lattice.

Lemma 4. *If* $A \in \overline{D}$ *and* $x, y \in A$ *are isolated, then* x *and* y *have an isolated upper bound* $z \in A$.

Proof. Let $A_0 \subseteq D$ be directed with $A_0^c = A$. Let $x = \bigvee X$ where X is directed and $\forall w \in X\ \exists a \in A_0\ w \leq a$. Since x is isolated, $x \in X$. Thus there exists $x' \in A_0$ with $x \leq x'$. Similarly there exists $y' \in A_0$ with $y \leq y'$. Let $z' \in A_0$ be an upper bound for x' and y'. Let z be an isolated upper bound for x and y with $z \leq z'$. $z \in A$ since $z' \in A$ and A is downward-closed. ∎

Lemma 5. *Let* $A \in \overline{D}$, *and let* A_0 *be the set of isolated elements of* A. A_0 *is directed, and* $A_0^c = A$.

Proof. Immediate from Lemma 4 and the fact that every element of D is a least upper bound of isolated elements. ∎

Theorem 6. *If* $X \subseteq \overline{D}$ *is directed, then* X *has a least upper bound in* $\overline{D}$ *given by*

$$\bigsqcup X = (\bigcup X)^c.$$

Proof. It suffices to show that $(\bigcup X)^c \in \overline{D}$, which requires finding a directed $Y \subseteq D$ with $Y^c = (\bigcup X)^c$.

For $A \in X$, let Y_A be the set of isolated elements of A. Each Y_A is directed, and $Y_A^c = A$.

Since X is directed in $\overline{D}$, $\bigcup_{A \in X} Y_A$ is directed in D. Clearly $\bigcup_{A \in X} Y_A \subseteq \bigcup X$, so

$$(\bigcup_{A \in X} Y_A)^c \subseteq (\bigcup X)^c.$$

Let $x \in (\bigcup X)^c$, with $x = \bigvee Z'$ where Z' is directed and $Z' \subseteq \bigcup X$. By Lemma 2 there exists a directed set $Z \subseteq \bigcup X$ consisting solely of isolated elements such that $x = \bigvee Z$. Since $Z \subseteq \bigcup_{A \in X} Y_A$, $x \in (\bigcup_{A \in X} Y_A)^c$.

Therefore $(\bigcup_{A \in X} Y_A)^c = (\bigcup X)^c$, and $(\bigcup X)^c \in \overline{D}$. ∎

Hence $\langle \overline{D}, \sqsubseteq \rangle$ is a complete partial order.

D may be regarded as a subset of $\overline{D}$ via the continuous injection $\bar{x}$ given by $x \mapsto \bar{x}$ where $\bar{x} = \{ d \in D \mid d \leq x \}$. From Lemma 5 it can be seen that

$$A = \bigsqcup_{\substack{x \in A \\ x \text{ isolated}}} \bar{x}$$

for any $A \in \overline{D}$. The following theorem thus completes a proof that $\overline{D}$ is an ω-complete domain with least element $\{ \perp \}$.

Theorem 7. *The isolated elements of $\overline{D}$ are precisely the images $\bar{x}$ of isolated elements x in D.*

Proof. Let x be isolated in D, and let $X \subseteq \overline{D}$ be directed with

$$\bar{x} = \{ y \mid y \leq x \} \sqsubseteq \bigsqcup X.$$

By Theorem 6 $x \in (\bigcup X)^c$, so let $Y \subseteq \bigcup X$ be directed with $x = \bigvee Y$. x is isolated, so $x \in Y$. Thus $x \in A$ for some $A \in X$. For that A, $\bar{x} \sqsubseteq A$. Therefore $\bar{x}$ is isolated in $\overline{D}$.

Conversely, let $A \in \overline{D}$ be isolated. Let $\{ \perp = b_0, b_1, b_2, \ldots \}$ be the isolated elements of D. Define an increasing sequence $\{ x_i \}_{i \in \omega}$ in D by

$$x_0 = b_0 = \perp$$
$$x_{i+1} = \begin{cases} x_i & b_{i+1} \notin A; \\ b_k & b_{i+1} \in A, \end{cases}$$

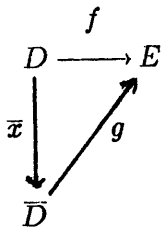
where $k = \mu n [b_{i+1} \leq b_n \ \wedge \ x_i \leq b_n \ \wedge \ b_n \in A]$. The isolated elements of A are directed, and $x_i \in A$, so k is defined whenever $b_{i+1} \in A$. For every isolated $y \in A$ there exists k such that $y \leq x_k$. Since $A = (\{ y \in A \mid y \text{ isolated} \})^c$, $A = \bigsqcup_{i \in \omega} \bar{x}_i$. $\{ \bar{x}_i \mid i \in \omega \}$ is directed since $\{ x_i \}$ is increasing. Therefore $A = \bar{x}_i$ for some i. ∎

That $\overline{D}$ is the unique completion of D is guaranteed by a universal mapping property.[13] This universal property is hardly more than a paraphrase of a theorem on finitary categories by Smyth and Plotkin.[14]

[13] Saunders MacLane, *Categories for the Working Mathematician*, Springer-Verlag, New York, 1971.

[14] M B Smyth and G D Plotkin, "The category-theoretic solution of recursive domain equations", Proceedings 18th Annual IEEE Symposium on Foundations of Computer Science, 1977, pages 13–17.

Theorem 8. *If E is an ω-complete domain, and $f:D \rightarrow E$ is ω-continuous, then there exists a unique ω-continuous map $g:\overline{D} \rightarrow E$ making the diagram below commute.*



In other words, any continuous map f from D to E factors uniquely through $\overline{D}$: $f = g \circ \bar{x}$. This means that any ω-complete domain containing D also contains $\overline{D}$, so that $\overline{D}$ is the smallest ω-completion of D. Furthermore any ω-complete domain with this property is isomorphic to $\overline{D}$ via a unique isomorphism.

III.4. The Power Domain

The idea of power domains is that a nondeterministic function may be described as a deterministic set-valued function, where the set contains all values the nondeterministic function can take for the given argument. Consider, for example, the program

```
     ∇ RESULT ← FOO X
[1]  GLOBAL ← 0
[2]  RESULT ← (F X) + (G X)
     ∇

     ∇ RESULT ← F X
[1]  RESULT ← GLOBAL
[2]  GLOBAL ← 1
     ∇

     ∇ RESULT ← G X
[1]  RESULT ← GLOBAL
[2]  GLOBAL ← 1
     ∇
```

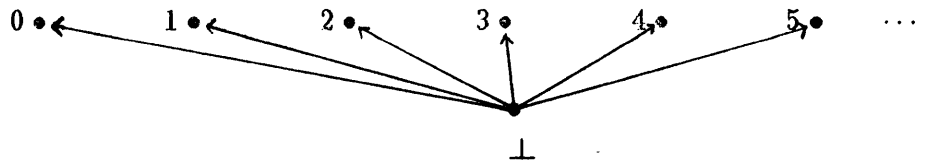


Figure 5. N, the flat domain of natural numbers.

defined in §1. When the subprograms F and G are evaluated in parallel on a multiprocessor, FOO can map its input to either 0 or 1. This behavior can be described by

$$x \mapsto \{0, 1\},$$

and this is the best description of FOO's input-output behavior possible when arguments are evaluated in parallel.

Since fixed point semantics works by generating a sequence of ever-better approximations to the meaning of a program, some ordering $\sqsubseteq$ must be placed on sets of values so that $A \sqsubseteq B$ means that B is at least as good an approximation as A. The values will be drawn from some domain $\langle D, \leq \rangle$.

One of the simplest domains is the flat domain of natural numbers $\langle \mathbf{N}, \leq \rangle$, where $\mathbf{N} = \{\perp, 0, 1, 2, 3, \ldots\}$ and $x \leq y$ iff $x = y$ or $x = \perp$. (Note that $\leq$ is not the usual ordering on N.) This domain is pictured in Figure 5. Suppose for simplicity that APL programs can return only nonnegative integers as values, so that the output of FOO lies in N. As already noted, the possible outputs of FOO when arguments are evaluated in parallel are best described by the set

$$\{0, 1\}.$$

Which subsets of N should count as approximations to this set? There are at least three reasonable answers. To each answer there corresponds a way of interpreting sets, and to each interpretation there corresponds a preorder. The three preorders we will consider are written $\sqsubseteq_0$, $\sqsubseteq_1$, and $\sqsubseteq_{\text{E-M}}$.

One approach is to interpret a set as including a description of every possible output value. Not every element of the set has to describe an output value, but every output value has to be described by an element of the set. In this approach N and $\{0, 1, 2\}$ both approximate $\{0, 1\}$, but $\{0, 1, 2\}$ is a more refined approximation than N:

$$\mathbf{N} \sqsubseteq_0 \{0, 1, 2\} \sqsubseteq_0 \{0, 1\}.$$

$\{0, 1, 3\}$ is an example of an approximation to $\{0, 1\}$ that is incomparable with $\{0, 1, 2\}$. For general domains this approximation ordering is given for $A, B \subseteq D$ by

$$A \sqsubseteq_0 B \quad \Leftrightarrow \quad \forall y \in B\, \exists x \in A\, x \leq y.$$

As is true also of the next two approximation orderings, $\sqsubseteq_0$ is in general only a preorder. In the case of $\sqsubseteq_0$, $\{\perp\} \sqsubseteq_0 D \sqsubseteq_0 \{\perp\}$. $\sqsubseteq_0$ is the *Smyth ordering*, and yields a so-called weak power domain.[15] It has been used to give a semantics for a model of concurrency based on message passing.[16]

Another approach is to interpret a set as giving descriptions of some possible output values. Not every possible output value has to be described by an element of the set, but every element of the set has to describe an output value. In this approach $\{\perp\}$ and $\{0\}$ both approximate $\{0, 1\}$, but $\{0\}$ is a more refined approximation that $\{\perp\}$:

$$\{\perp\} \sqsubseteq_1 \{0\} \sqsubseteq_1 \{0, 1\}.$$

$\{1\}$ is an example of an approximation to $\{0, 1\}$ that is incomparable with $\{0\}$. For general domains this approximation ordering is given for $A, B \subseteq D$ by

$$A \sqsubseteq_1 B \quad \Leftrightarrow \quad \forall x \in A\, \exists y \in B\, x \leq y.$$

In this ordering approximations build up to a limit, while in the Smyth ordering approximations narrow down to a limit. In other words, $\sqsubseteq_1$ corresponds to a generative approach while $\sqsubseteq_0$ corresponds to a restrictive approach. $\sqsubseteq_1$ also gives rise to a weak power domain, and has been used in the theory of Petri nets.[17] The actor semantics presented in the next chapter will use $\sqsubseteq_1$.

Historically, the first approach was to interpret a set in both of the preceding ways. For flat domains such as N the *Egli-Milner* ordering $\sqsubseteq_{\text{E-M}}$ was defined by

$$A \sqsubseteq_{\text{E-M}} B \quad \Leftrightarrow \quad (\perp \notin A \wedge A = B) \vee (\perp \in A \wedge (A - \{\perp\}) \subseteq B).$$

[15]M B Smyth, "Power domains".

[16]George Milne and Robin Milner, "Concurrent processes and their syntax", *JACM* 26, 2, April 1979, pages 302–321.

[17]Mogens Nielsen, Gordon Plotkin, and Glynn Winskel, "Petri nets, event structures and domains", in *Semantics of Concurrent Commputation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979, pages 266–284.

Gordon Plotkin generalized to arbitrary domains by the definition[18]

$$A \sqsubseteq_{\text{E-M}} B \quad \Leftrightarrow \quad A \sqsubseteq_0 B \wedge A \sqsubseteq_1 B.$$

In this approach $\{\perp\}$ and $\{\perp, 0\}$ both approximate $\{0, 1\}$, but $\{\perp, 0\}$ is a more refined approximation than $\{\perp\}$:

$$\{\perp\} \sqsubseteq_{\text{E-M}} \{\perp, 0\} \sqsubseteq_{\text{E-M}} \{0, 1\}.$$

$\{\perp, 1\}$ is an example of an approximation to $\{0, 1\}$ that is incomparable with $\{\perp, 0\}$. The Egli-Milner ordering has been used to give a semantics for Communicating Sequential Processes, a language based on message passing.[19]

Each of the three preorders, $\sqsubseteq_0$, $\sqsubseteq_1$, and $\sqsubseteq_{\text{E-M}}$, gives rise to a power domain construction applicable to any ω-complete partial order having a least element.[20] But for the need to solve recursive domain equations involving power domains, at least the first two of these constructions could be extended to incomplete domains as well. In the actor semantics presented in Chapter IV there is no need to solve recursive domain equations involving the power domain. Furthermore the domain of augmented event diagrams, from which the actor power domain is to be built, is naturally incomplete. The remainder of this section therefore defines power domains for all domains, complete or incomplete, and shows that for an incomplete domain D the power domain so defined is isomorphic to the conventionally defined power domain of its ω-completion $\overline{D}$.

Michael Smyth has given a succinct characterization of conventional power domains, which we will now review.[21] He points out that the simplest way to build a power domain is first to decide what is to count as a finite piece of information about the result of a computation, and then to place an approximation ordering on the finite pieces of information. The power domain then becomes the essentially unique completion of the partial order so defined.

[18]G D Plotkin, "A powerdomain construction", *SIAM J Computing* **5**, 3, September 1976, pages 452–487.

[19]Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, 1979, pages 290–308.

[20]M C B Hennessy and G D Plotkin, "Full abstraction for a simple parallel programming language", FOCS-79, Springer-Verlag Lecture Notes in Computer Science 74, 1979.

[21]M B Smyth, "Power domains."

Let $\langle D, \leq \rangle$ be a domain. In the most commonly encountered domains, isolated elements represent finite chunks of information in D, and indeed the term "finite" is often used in place of "isolated". A finite piece of information should therefore be a nonempty finite set of isolated elements from D. Smyth preordered these sets using $\sqsubseteq_0$ and $\sqsubseteq_{\text{E-M}}$, but we will use $\sqsubseteq_1$, so that a nonempty finite set of isolated elements $A \subseteq D$ is interpreted to mean

$$\forall a \in A\ \exists r \in R\ a \leq r$$

where R is the actual set of values possible as the result of a nondeterministic program. Letting $A =_1 B$ iff $A \sqsubseteq_1 B$ and $B \sqsubseteq_1 A$, the equivalence classes of such sets under $=_1$ are partially ordered by the quotient ordering $\sqsubseteq_1 / =_1$.

The equivalence classes can be avoided by dealing with distinguished representatives of them. Accordingly define the *finite frontiers* of D as

$$F(D) = \{ A \subseteq D \mid A \text{ is a nonempty finite set of isolated elements, and } \forall x, y \in A\ x \leq y \Rightarrow x = y \}.$$

$A \in F(D)$ is called a frontier because each of its elements is both minimal and maximal in A. $\langle F(D), \sqsubseteq_1 \rangle$ is isomorphic to the set of equivalence classes under $=_1$ of nonempty finite sets of isolated elements of D, ordered by $\sqsubseteq_1 / =_1$. $\langle F(D), \sqsubseteq_1 \rangle$ is a domain with least element $\{ \bot \}$ in which every element is isolated. It therefore has an ω-completion $\langle \overline{F(D)}, \sqsubseteq \rangle$, which is the power domain, up to isomorphism.

Observe that only the isolated elements of D matter to the construction. It is therefore irrelevant whether D is ω-complete.

The following lemma characterizes the conventional power domain $\langle \overline{F(D)}, \sqsubseteq \rangle$.

Lemma 1. *$S \in \overline{F(D)}$ if and only if both the following hold:*

1. $S = \{ F \in F(D) \mid F \subseteq \bigcup S \}$.

2. If $s \in \bigcup S$, x is isolated, and $x \leq s$, then $x \in \bigcup S$.

Proof. Since every element of $F(D)$ is isolated, $S \in \overline{F(D)}$ iff S is downward-closed and directed as a subset of $F(D)$.

Let $S \in \overline{F(D)}$.

Let $s \in \bigcup S$, and let x be isolated with $x \leq s$. There exists $F \in S$ with $s \in F$, and so by definition $\{x\} \sqsubseteq_1 F$. Therefore $\{x\} \in S$ since S is downward-closed, whence $x \in \bigcup S$.

If $F \in S$, then $F \in F(D)$ and $F \subseteq \bigcup S$. Suppose $F = \{f_0, \ldots, f_n\} \in F(D)$ with $F \subseteq \bigcup S$. Since S is downward-closed, $\{f_i\} \in S$ for $i = 0, \ldots, n$. Since S is directed, $F = \bigsqcup_1 \{\{f_i\} \mid i = 0, \ldots, n\} \in S$.

Conversely, suppose S satisfies conditions 1 and 2 of the lemma. Let $F_1, F_2 \in S$. $F_1 \sqcup_1 F_2 \subseteq F_1 \bigcup F_2 \subseteq \bigcup S$, so $F_1 \sqcup_1 F_2 \in S$ and S is directed. Let $F_1 \in S$, and let $F \sqsubseteq_1 F_1$. For every $x \in F$ there exists $s \in F_1$ with $x \leq s$. By condition 2 $F \subseteq \bigcup S$. By condition 1 $F \in S$. Thus S is downward-closed. Being downward-closed and directed, $S \in \overline{F(D)}$. ∎

A corollary of this lemma is that the least upper bound of an increasing sequence $\{S_i\}_{i \in \omega}$ in $\overline{F(D)}$ is given by $\bigsqcup_{i \in \omega} S_i = \bigcup_{i \in \omega} S_i$.

The concrete power domain that we will use is defined below. As will be shown, it is isomorphic to $\langle \overline{F(D)}, \sqsubseteq \rangle$.

The closure operation ^{c} was defined in §3.

Definition. *Let* $\langle D, \leq \rangle$ *be a domain. Its* power domain *is* $\langle P[D], \sqsubseteq \rangle$, *where*

$$P[D] = \{A^c \mid \perp \in A \subseteq D\}$$

and, for $A, B \in P[D]$,

$$A \sqsubseteq B \Leftrightarrow A \subseteq B.$$

In other words, $P[D]$ is the collection of downward-closed subsets of D that are also closed under existing least upper bounds of directed sets in D. Note that while the ordering on $P[D]$ is given by the subset relation, least upper bounds do not in general coincide with unions.

For the actor event diagram domain $\mathbf{D}$, an element of $P[\mathbf{D}]$ represents a list of possible initial histories of a computation. Since for elements x and y of $\mathbf{D}$, $x \leq y$ means that x is an initial segment of the initial history y, the requirement that elements of $P[\mathbf{D}]$ be downward-closed has a clear basis in intuition.

The next theorem gives several nice properties of the power domain. In particular, it is an ω-complete domain, so ω-continuous functions have fixed points.

Definition. *A* countably based continuous complete lattice *is an ω-complete domain such that for any subset X of the domain both a least upper bound $\bigsqcup X$ and a greatest lower bound $\sqcap X$ exists.*

Theorem 2. *If $\langle D, \leq \rangle$ is a domain, then $\langle P[D], \sqsubseteq \rangle$ is a countably based continuous complete lattice.*

Proof. If $X \subseteq P[D]$, then $\bigsqcup X = (\bigcup X)^c$ and $\sqcap X = \bigcap X$.

The isolated elements of $P[D]$ are the closures of finite sets of isolated elements, that is sets of the form $\{ x_0, \ldots, x_n \}^c$ where $x_0, \ldots, x_n$ are isolated in D. To prove it, let $x_0, \ldots, x_n$ be isolated and let $X \subseteq P[D]$ be directed with $\{ x_0, \ldots, x_n \}^c \sqsubseteq \bigsqcup X$. Since $x_i \in (\bigcup X)^c$ and x_i is isolated, $x_i \in \bigcup X$. Let $A_i \in X$ have x_i as an element. Let $A \in X$ be an upper bound for $A_0, \ldots, A_n$. $\{ x_0, \ldots, x_n \}^c \sqsubseteq A$.

Conversely, let $A \in P[D]$ be isolated and let $\{ x_i \mid i \in \omega \}$ be the isolated elements of A. Let

$$X_n = \{ x_i \mid i \leq n \}.$$

Then $\{ X_n^c \}_{n \in \omega}$ is an increasing sequence in $P[D]$ and $A = \bigsqcup_{n \in \omega} X_n^c$, so for some n, $A = X_n^c$. ∎

The following theorem says that at a certain level of abstraction $P[D]$ is the same as the conventional power domain of D. While $P[D]$ will be used in the next chapter to give a semantics for actor-based programming languages with unbounded nondeterminism, however, the conventional power domain is usually considered incapable of expressing unbounded nondeterminism. This points out the importance of the concrete interpretation placed upon elements of the power domain.

Theorem 3. *If $\langle D, \leq \rangle$ is a domain, then $\langle P[D], \sqsubseteq \rangle$ is isomorphic to $\langle \overline{F(D)}, \sqsubseteq \rangle$.*

Proof. Consider the map from $F(D)$ to $P[D]$ given by $F \mapsto F^c$. This map is monotonic and is trivially continuous since $F(D)$ has only isolated elements. By Theorem 8 of §3 there exists a unique continuous extension of this map with domain $\overline{F(D)}$. This unique extension is η: $\overline{F(D)} \to P[D]$ with

$\eta(S) = (\bigcup S)^c$ for all $S \in \overline{F(D)}$. It remains to be shown that η is one-to-one and onto and has a continuous inverse.

$(\bigcup S)^c$ is the same as $(\bigcup S)$ except for non-isolated elements, so η is one-to-one by Lemma 1. If $A \in P[D]$, then

$$\begin{aligned} A &= \{ x \in A \mid x \text{ is isolated} \}^c \\ &= \eta(\{F \in F(D) \mid F \subseteq A\}) \end{aligned}$$

so η is onto $P[D]$.

The inverse of η is θ: $P[D] \to \overline{F(D)}$ where $\theta(A) = \{F \in F(D) \mid F \subseteq A\}$. θ is clearly monotonic. To show θ continuous, let $\{A_i\}_{i\in\omega}$ be an increasing sequence in $P[D]$ and let $F \in \theta(\bigsqcup_{i\in\omega} A_i)$. That is, $F \subseteq \bigsqcup_{i\in\omega} A_i = (\bigcup_{i\in\omega} A_i)^c$. Each $x \in F$ is isolated and so $x \in A_i$ for some i. F is a finite set, and $\{A_i\}_{i\in\omega}$ is an increasing sequence, so $F \subseteq A_i$ for some i. Therefore $F \in \bigsqcup_{i\in\omega} \theta(A_i)$. ∎

III.5. Power Domains from Incomplete Domains

Usually the partial order from which the power domain is constructed is required to be ω-complete. There are two reasons for this. The first reason is that most power domains are simply generalizations of domains that have been used as semantic domains for conventional sequential programs, and such domains are all complete because of the need to compute fixed points in the sequential case. The second reason is that ω-completeness permits the solution of recursive domain equations involving the power domain such as

$$R \simeq S \to P[S + (S \times R)]$$

which defines a domain of resumptions.[22] As shown in the previous section, however, power domains can be defined for any domain whatsoever. Furthermore the power domain of a domain is essentially the power domain of its ω-completion, so recursive equations involving the power domain of an incomplete domain can still be solved, provided the domains to which the usual constructors ($+$, $\times$, $\to$, and *) are applied are ω-complete. It happens that defining actor semantics as in the next chapter does not require solving any recursive equations involving the power domain.

[22] Plotkin, "A powerdomain construction".

In short, there is no technical impediment to building power domains from incomplete domains. But why should one want to do so?

In *behavioral semantics*, developed by Irene Greif, the meaning of a program is a specification of the computations that may be performed by the program. The computations are represented formally by the actor event diagrams considered in Chapter II. Greif specified the event diagrams by means of *causal axioms* governing the behaviors of individual actors.[23]

Henry Baker has presented a nondeterministic interpreter generating instantaneous schedules which then map onto event diagrams. He suggested that a corresponding deterministic interpreter operating on sets of instantaneous schedules could be defined using power domain semantics.[24]

The semantics presented in the next chapter is a version of behavioral semantics. A program will denote a set of actor event diagrams. That set will be defined extensionally using power domain semantics rather than intensionally using causal axioms. The behaviors of individual actors will be defined functionally. It will be shown, however, that the resulting set of actor event diagrams consists of exactly those diagrams that satisfy causal axioms expressing the functional behaviors of actors. Thus Greif's behavioral semantics is compatible with a denotational power domain semantics.

Baker's instantaneous schedules introduced the notion of *pending events*, which represent messages on the way to their targets or in the process of being sent. Each pending event must become an actual (realized) event sooner or later, a requirement referred to as *finite delay*. Augmenting actor event diagrams with sets of pending events helps to express the finite delay property, which is characteristic of true concurrency.[25]

The augmented actor event diagrams form a partially ordered set $\langle \mathbf{D}, \leq \rangle$ from which to construct the power domain $P[\mathbf{D}]$. The augmented diagrams are partial computation histories representing "snapshots" of a computation on its way to being completed. For $x, y \in \mathbf{D}$, $x \leq y$ means x is a stage the computation could go through on its way to y. The completed elements of $\mathbf{D}$ represent computations that have terminated and nonterminating computations that have become infinite. The

[23]Irene Greif, "Semantics of communicating parallel processes", MIT Project MAC Technical Report 154, September 1975.

[24]Henry Baker, "Actor systems for real-time computation", MIT LCS Technical Report 197, March 1978.

[25]Jerald S Schwarz, "Denotational semantics of parallelism", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.

completed elements may be characterized abstractly as the maximal elements of **D**.[26] Concretely, the completed elements are those having no pending events. Intuitively, **D** is not ω-complete because there exist increasing sequences of finite partial computations

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq \cdots$$

in which some pending event remains pending forever while the number of realized events grows without bound, contrary to the requirement of finite delay. Such a sequence cannot have a limit, because any limit would represent a completed nonterminating computation in which an event is still pending.

Many readers will be concerned about the possibility of a nonterminating computation proceeding merrily along from one finite stage to the next but blowing up at infinity without a trace, that is, without an element in **D** to represent the entire nonterminating computation. That cannot happen. In Chapter IV it will be shown for every program that the set of partial computations that can occur is exactly the set of initial histories of the completed computations that can occur. Every element of **D** lies below a completed element, and the completed elements represent all possible completed computations, both terminating and nonterminating. If an increasing sequence does not have a limit, then it does not represent a possible computation, because the sequence reveals a message that is sent but that never arrives at its target. ω-incompleteness thus follows from the assumption of finite delay.

The fact that there exist increasing sequences without least upper bounds will seem strange to those accustomed to thinking about the semantics of sequential programs. It may help to point out that the increasing sequences produced by sequential programs all have least upper bounds. Indeed, the partial computations that can be produced by sequential computations form an ω-complete subdomain of **D**. An informal proof follows.

From the actor point of view, sequential computations are a special case of concurrent computations, distinguishable by their event diagrams. The event diagram of a sequential computation has an initial event, and no event activates more than one event. In other words, the activation ordering of a sequential computation is linear; the event diagram is essentially a conventional execution sequence.

[26] See §0 of William W Wadge, "An extensional treatment of dataflow deadlock", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.

This means that the finite elements of **D**

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq \cdots$$

corresponding to the finite initial segments of a sequential execution sequence all have exactly one pending event, excepting the largest, completed element if the computation terminates. One property of the augmented event diagrams domain $\langle \mathbf{D}, \leq \rangle$ is that if $x \leq y$ and $x \neq y$, then some pending event of x is realized in y. Since in this case each x_i has at most one pending event, every pending event in the sequence becomes realized. Hence the sequence

$$x_0 \leq x_1 \leq x_2 \leq x_3 \leq \cdots$$

has a least upper bound in **D**, in accord with intuition.

The above proof applies to all sequential programs, even those with choice points such as guarded commands. Thus actor semantics includes sequential programs as a special case, and agrees with conventional semantics on the meanings of such programs.

For convenience, though, the behavioral semantics presented in the next chapter will assume that all actors are deterministic, which rules out choice points. We exclude choice nondeterminism, the better to study arrival nondeterminism.

To repeat, the actor event diagram domain **D** is incomplete because of the requirement of finite delay, which allows any finite delay between an event and an event it activates but rules out infinite delay. Finite delay follows from leaving much timing information unspecified, such as the cylinder that happens to be under a disk head at a particular instant, the detailed time-dependent behavior of a communications network, the relative speeds of concurrent processors, and the exact times at which inputs are presented to the computing system by the external world. All these timing details are suppressed in the interest of obtaining greater abstraction.

The next three sections explain the relation between finite delay and fair parallelism.

III.6. Implementations are not Meanings

> It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct.

Thus spoke Dana Scott of the purposes of a programming language semantics.[27] Usually, however, the formal semantics of a conventional sequential programming language may itself be interpreted to provide an (inefficient) implementation of the language. A formal semantics need not always provide such an implementation, though, and to believe that semantics must provide an implementation leads to confusion about the formal semantics of nondeterministic languages. Such confusion is painfully evident when the presence of unbounded nondeterminism in a programming language's semantics is said to imply that the programming language cannot be implemented.

Although the meaning of a computer program may be described by an element of a power domain, so that the program's meaning is a set, execution of the program is not supposed to produce the set as its answer. Rather the set describes the possible outcomes of executing the program.

Indeed, although the meaning of the program is represented as a set of possible outcomes, it is not necessary that every possible outcome be possible in every implementation of the program. This permits nondeterministic languages to be implemented efficiently on deterministic, sequential machines.

In other words, implementations are not required to preserve all the nondeterminism present in the semantics. This corresponds to *loose nondeterminism* in the distinction drawn by David Park:[28]

> *tight nondeterminism:* each correct implementation must, according to some precise sense of "possible result", produce all and only those possible results which the semantics of the language prescribes.
>
> *loose nondeterminism:* there may or may not be a sense in which the implementation can produce more than one result; the only constraint is that every result produced is one of those prescribed by the semantics.

III.7. Choice Nondeterminism is Bounded

Unbounded nondeterminism, defined below, is an arcane technical notion of little interest in its own right. It is useful in pointing out the difference between choice nondeterminism and the

[27] "What is Denotational Semantics?", MIT Laboratory for Computer Science Distinguished Lecture Series, 17 April 1980.

[28] "On the semantics of fair parallelism", University of Warwick Theory of Computation Report 31, October 1979.

nondeterminism that arises from concurrency, and in discussing the interesting and practical question of fairness.

If, for some fixed input, a program always returns an answer but the number of possible answers is infinite, then the program is said to exhibit *unbounded nondeterminism.* Unbounded nondeterminism as thus defined is not a very precise concept since it depends critically upon the meaning of "possible". In my opinion it is best to take the possible answers as those permitted by the semantics of the programming language in which the program is written. This gives unbounded nondeterminism a meaning as precise as can be had given the semantics of the language under consideration. Under this interpretation unbounded nondeterminism is a property of programs, not a property of implementations.

Nondeterminism that is not unbounded is *bounded.* Thus the nondeterminism of a program that may not halt is bounded.

Nondeterministic Turing machines have only bounded nondeterminism.[29] Sequential programs containing guarded commands as the only sources of nondeterminism have only bounded nondeterminism.[30] Briefly, choice nondeterminism is bounded. Plotkin gave a proof in his original paper on power domains:[31]

> Now the set of all initial segments of execution sequences of a given nondeterministic program P, starting from a given state, will form a tree. The branching points will correspond to the choice points in the program. Since there are always only finitely many alternatives at each such choice point, the branching factor of the tree is always finite. That is, the tree is finitary. Now Konig's lemma says that if every branch of a finitary tree is finite, then so is the tree itself. In the present case this means that if every execution sequence of P terminates, then there are only finitely many execution sequences. So if an output set of P is infinite it must contain [a nonterminating computation].

This proof depends upon the premise that if every node x of a certain infinite branch can be reached by some computation c, then there exists a computation c that goes through every node x on the

[29]A nondeterministic Turing machine is a mathematical abstraction, not a physical machine. A given nondeterministic Turing machine is thus better viewed as a program than as an implementation.

[30]Edsger Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

[31]G D Plotkin, "A powerdomain construction", *SIAM J Computing* 5, 3, September 1976, pages 452–487.

branch. In other words, the premise is of the form

$$\forall x\, \exists c\, F(x, c) \Rightarrow \exists c\, \forall x\, F(x, c).$$

Clearly this premise follows not from logic but rather from the interpretation given to choice points. This premise fails for arrival nondeterminism because of finite delay. Though each node on an infinite branch must lie on a branch with a limit, the infinite branch need not itself have a limit. Thus the existence of an infinite branch does not necessarily imply a nonterminating computation.

The following program, written in Communicating Sequential Processes,[32] is an example of a program with choice nondeterminism. Its nondeterminism is therefore bounded.

```
[P :: n: integer; n := 0;
      guard: boolean; guard := true;
      *[guard → n := n + 1
        □ guard → guard := false]
]
```

The repetitive guarded command might never terminate, because the first guard might always be chosen in preference to the second. While in a sense this is unfair to the second guard, it is allowed by the interpretation of choice points, because random choice is a valid implementation of choice points. An implementation that chose guards at random might choose the first guard on each repetition, and while the probability of that happening would be zero it would still be possible. Since the implementation using random choice is allowed to choose the first guard forever, deterministic implementations are also allowed to choose the first guard forever. According to loose nondeterminism, therefore, in some valid implementations this program could not possibly halt.

Arrival nondeterminism, however, can be bounded. Consider a dual processor system. As timesharing users know, from a user's viewpoint the effective speed of a processor varies with the computational tasks it is called upon to perform. Suppose one of the dual processors is used for timesharing as well as batch computation while the other is reserved for batch computation. As the

[32] C A R Hoare, "Communicating sequential processes", *CACM* 21, 8, August 1978, pages 666-677.

timesharing load increases, the relative effective speeds of the two processors varies. The effective speed ratio is bounded only by the degraded response time that users are willing to tolerate, so for the purposes of mathematical discussion the effective speed ratio is unbounded.

The unboundedness of the effective speed ratio gives rise to unboundedly nondeterministic programs. Suppose the timesharing processor counts to 100 and then sends a message to the other processor. Meanwhile the relatively free processor has been counting as fast as it can; how high can it count before it receives the message? As more users burden the timesharing processor, successive runs of the program yield higher and higher counts. No principled bound can be set.

One possible objection to this scenario as an example of unbounded nondeterminism is that the behaviors of the timesharing users and the timesharing system must be included in any proper account of the concurrent counting program. If this objection is to be allowed, though, the semantics of concurrent programs becomes quite intractable. Semantics is useful only to the extent that such details can be suppressed.

An analogous scenario can be constructed for a single sequential machine through the use of two agendas from which tasks are selected alternately and to which tasks are added unevenly. Again an unbounded delay can be achieved. It is the property of finite but unbounded delay that gives rise to unbounded nondeterminism. Finite delay is a common and natural property of abstract descriptions of concurrent systems.

III.8. Fairness Implies Unbounded Nondeterminism

Fairness, roughly speaking, is a property of programs that take inputs from two or more concurrent processes in such a way that each attempt by a process to provide input is bound to succeed sooner or later. A fair (two-way) merge, for example, is a program that takes values produced by two processes and merges them into a single sequence, never ignoring forever a value that one of the processes is trying to feed it. If one of the processes generates an infinite sequence of zeroes and the other an infinite sequence of ones, then the set of sequences that could be produced by a fair merge of those processes is the set of sequences containing infinitely many zeroes, infinitely many ones, and

nothing else; formally

$$(0^*11^*0)^\omega.$$

An unfair merge would be a sequence with only finitely many zeroes or ones.

The ability to write a fair merge is very important to programmers of operating systems and concurrent systems. By no means is it an ability provided by all concurrent programming languages. Unbounded nondeterminism serves as one test for fairness: if a fair merge can be written in the language, then the fair merge can be used to write a program with unbounded nondeterminism. To see the idea behind this bit of folk wisdom, consider a program written in Communicating Sequential Processes (CSP):[33]

```
[X :: Z!stop() ||

 Y :: guard: boolean; guard := true;
      *[guard → Z!go(); Z?guard] ||

 Z :: n: integer; n := 0;
      continue: boolean; continue := true;
      *[X?stop() → continue := false
        □ Y?go() → n := n + 1; Y!continue]
]
```

This program illustrates global nondeterminism, since the nondeterminism arises from incomplete specification of the timing of signals between the three processes X, Y, and Z. The repetitive guarded command in the definition of Z has two alternatives: either the `stop` message is accepted from X, in which case *continue* is set to `false`, or a `go` message is accepted from Y, in which case n is incremented and Y is sent the value of *continue*. If Z ever accepts the `stop` message from X, then X terminates. Accepting the `stop` message causes *continue* to be set to `false`, so after Y sends its next `go` message Y will receive `false` as the value of its guard and will terminate. When both X and Y have terminated, Z terminates because it no longer has live processes providing input.

[33]C A R Hoare, "Communicating sequential processes", *CACM* 21, 8, August 1978, pages 666-677.

As the author of CSP points out, therefore, if the repetitive guarded command in the definition of Z were required to be fair, this program would have unbounded nondeterminism: it would be guaranteed to halt but there would be no bound on the final value of n. In actual fact, the repetitive guarded commands of CSP are not required to be fair, and so the program may not halt.[34] This fact may be confirmed by a tedious calculation using the semantics of CSP,[35] or simply by noting that the semantics of CSP is based upon a conventional power domain and thus does not give rise to unbounded nondeterminism.

The reason unbounded nondeterminism does not apear in conventional power domain semantics is that each element of the power domain is interpreted as a finitely generable subset of the underlying ω-complete domain. In the ω-complete domains that have been proposed, finitely generable subsets are either finite or contain an element representing a nonterminating or undefined computation, for essentially the same reason that choice nondeterminism is bounded.[36] In the actor event diagram domain and its completion, however, the augmented diagrams contain so much operational information that one can distinguish computations that violate finite delay from other nonterminating computations. Intuitively, the actor event diagram domain is incomplete because the computations that violate finite delay have been thrown out.

To return to the proof that choice nondeterminism is bounded and to see why that proof does not work for arrival nondeterminism, it is first of all not clear that the tree of initial segments of execution sequences of a concurrent program is always finitary, since the alternatives may for example correspond to the wait times allowed by finite delay.[37] Secondly, an infinite branch does not necessarily indicate a nonterminating computation, since the path may violate the requirement of finite delay and thus not have a limit. Recall the fair merge of an infinite sequence of zeroes and an infinite sequence of ones. Every finite sequence of zeroes is a possible initial segment of a fair merge but the

[34] *ibid.*

[35] Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, 1979, pages 290–308.

[36] G D Plotkin, "A powerdomain construction", *SIAM J Computing* **5**, 3, September 1976, pages 452–487.

[37] Nancy A Lynch and Michael J Fischer, "On describing the behavior and implementation of distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979. See also R J Back, "Semantics of unbounded nondeterminism", Mathematisch Centrum Report IW 135/80, April 1980.

limit, an infinite sequence of zeroes, is impossible.

Apparently the designer of CSP stopped short of requiring fairness because at the time languages with unbounded nondeterminism were widely regarded as unimplementable.[38] Additionally unbounded nondeterminism would have precluded giving a conventional power domain semantics for CSP.

Another important proposal, based like CSP on message passing but more abstract than a programming language, is Concurrent Processes.[39] The semantics of Concurrent Processes also uses conventional power domains, so there is no unbounded nondeterminism and a fair merge cannot be specified.

It appears that a fair merge cannot be written as a nondeterministic data flow program operating on streams.[40] The reason is that for any monotonic function

$$merge\colon S \times S \rightarrow P[S]$$

from pairs of input streams to sets of possible output streams it must be that

$$merge(\perp, 1^\omega) \sqsubseteq merge(0, 1^\omega)$$

where $\perp$ is the empty stream. Since the only fair merge of $\perp$ and 1^ω is 1^ω, 1^ω should be an element of $merge(\perp, 1^\omega)$, but that would mean 1^ω must be an element of $merge(0, 1^\omega)$ also.

The coroutine proposal of Kahn and McQueen avoids nondeterminism altogether and thus cannot provide a fair merge. The "fair merge" that they present must assume for its correctness that both of its input streams are infinite.[41]

[38]"Communicating sequential processes".

[39]George Milne and Robin Milner, "Concurrent processes and their syntax", *JACM* 26, 2, April 1979, pages 302–321.

[40]Despite a claim to the contrary in Paul Roman Kosinski, "Denotational semantics of determinate and non-determinate data flow programs", MIT LCS Technical Report 220, May 1979. The proof of Theorem 5.2 in that paper mistakenly assumes trichotomy for partial orders. In fact the domain of tagged-stream-sets is incomplete, and the fixed points being manipulated in the remainder of that paper do not exist.

[41]Gilles Kahn and David McQueen, "Coroutines and networks of parallel processes", IFIP-77, Montreal, August 1977, pages 993–998.

It is possible to write a CSP program that acts as a fair two-way merge so long as neither process transmits infinitely many messages to it. Since CSP's semantics identifies all nonterminating computations, it is impossible to tell directly from the semantics whether the program is unfair in the infinite case. Since no CSP program has unbounded nondeterminism, however, one can conclude that writing a fair merge in CSP is impossible. In this way unbounded nondeterminism provides an indirect answer to the question of fairness even though the question cannot be formulated directly.

Notice in the context of loose nondeterminism that even though writing a fair merge in a given language may be impossible it may still be possible to write merge programs in the language that will in practice be implemented fairly. Indeed, the author of CSP has set forth the informal requirement that "an efficient implementation should try to be reasonably fair".[42] In practice implementations can be extremely fair. The fact that examination of a programming language's semantics shows that a fair merge cannot be written in the language reveals a deficiency not of the language but of the current theory of programming language semantics.

To sum up, the problem with choice points as a model of nondeterministic concurrency is that they cannot be used to write a fair merge. In terms of what programs can express about their implementations, merge programs using choice points can allow fair merge but they cannot require it.

How important is fairness? Every finite initial sequence of values produced by an unfair merge can also be produced by a fair merge. Fair and unfair merges differ only at infinity. It can be argued that fairness is therefore unimportant, since as finite beings our horizon of interest seldom extends beyond a few score billion years. This argument should appeal to those who for the same reason find silly the question of whether a program terminates or not.

[42]"Communicating sequential processes".

Chapter IV

Actor Semantics

This chapter sets forth a power domain semantics for actor-based languages. The semantics given here is a power domain formulation of the behavioral semantics invented by Irene Greif.[1] The semantics has an operational flavor because it gives as the meaning of a program a set of generalized execution sequences, which are essentially the actor event diagrams of Chapter II.

IV.1. Primitive Serializers

A *primitive serializer* is a special kind of actor. Conceptually a primitive serializer consists of an arbiter, a queue, and a processor. A primitive serializer is the target of an event when a message *arrives* at the serializer's arbiter and is placed in the serializer's queue to await processing. When two messages arrive at about the same time, the arbiter decides which one goes first in the queue. The arbiter must be reliable and place every incoming message in the queue. In other words, the arbiter performs a fair merge on incoming messages. The processor of the primitive serializer *accepts* messages serially from the queue and processes them according to some deterministic and terminating

[1]Irene Greif, "Semantics of communicating parallel processes", MIT Project MAC Technical Report 154, September 1975.

```
(stack = elements initially [ ]
           inside
         accept [ continuation op x ]
         if equal [ op "push" ]
            then change elements to [ x elements ] ;
                 send "pushed" to continuation
            else
         if equal [ op "pop" ]
            then
              (if equal [ elements [ ] ]
                   then send "error -- stack empty" to continuation
                   else change elements to second(elements) ;
                        send "popped" to continuation)
            else
         if equal [ op "top" ]
            then
              (if equal [ elements [ ] ]
                   then send "error -- stack empty" to continuation
                   else send first(elements) to continuation)
            else
         if equal [ op "empty?" ]
            then send equal [ elements [ ] ] to continuation
            else
         send "error -- undefined operation on stack" to continuation)
```

Figure 1. An implementation of a single stack in the toy programming language Atolia.

algorithm. When the processor accepts a message from the queue, it *locks* and accepts no more messages from the queue until it finishes with that message.

Messages are accepted and processed in the same order that they arrive at the primitive serializer, that is, in the same order as the arrival ordering of their corresponding events. Processing a message may involve (1) changing the *local state* of the primitive serializer's processor; (2) sending out a finite set of messages; (3) creating a finite set of new primitive serializers; this last possibility resembles process creation. When the processor finishes processing a message, it *unlocks* and accepts the next message in the queue. If there are none, it waits until there are.

Primitive serializers have been proposed as a basis for programming concurrent and distributed systems.[2] Figure 1, for example, shows one way to implement a stack as a primitive serializer. There is one state variable, `elements`, which is the empty sequence initially. `stack` takes messages of the form

[2]Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying and proving properties of guardians for distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.

[*continuation op x*]

where *continuation* is an actor that should receive the result or notification, *op* is one of the four stack operations (`push`, `pop`, `top`, and `empty?`), and x is a value to be pushed. When the operation is `pop`, `top`, or `empty?`, x may be omitted. The messages sent and the changes made to the local state variable should be apparent from the code. `stack` never creates any actors.

Programming languages based on primitive serializers, such as Act1[3] and Atolia,[4] will be called actor-based languages. Programs in such languages are often written in the object oriented, continuation passing style illustrated by `stack`.

IV.2. Actor Behaviors

For simplicity, this chapter will ignore actor creation. Chapter V will outline the small changes to the semantics given here that are necessary when actors can be created in the course of computation.

Let A be the set of actors, and M the set of messages.

An actor is completely described by its name and by its *behavior*, which specifies what the actor does whenever it receives a message. An actor's name is a necessary part of its description because two different actors may have the same behavior. An actor's behavior is a necessary part of its description because the same actor may have different behaviors at different times.

When a primitive serializer receives a message, it may change state, may send out a finite number of messages to other actors, and may create other actors. Ignoring the possibility of creating other actors, this suggests that the behavior of a primitive serializer a should be a function

$$b^a: \Sigma^a \rightarrow [\mathrm{M} \rightarrow (\Sigma^a \times (\mathrm{A} \times \mathrm{M})^*)],$$

where Σ^a is the set of *local states* of a, and an element of $(\mathrm{A} \times \mathrm{M})^*$ represents a finite set of messages sent out to specific targets. Since the only purpose of local states is to index the next behavior, though,

[3] *ibid.*

[4] See §V.5.

it is better to define the behavior domain F via the reflexive domain equation[5]

$$\mathbf{F} \simeq [\mathbf{M} \to (\mathbf{F} \times (\mathbf{A} \times \mathbf{M})^*)].$$

F may be thought of as the set of trees of height ω, with an unlabelled root node, non-root nodes labelled by finite sequences in $\mathbf{A} \times \mathbf{M}$, and such that each node has exactly one outgoing arc labelled by m for each message $m \in \mathbf{M}$.

Behaviors are normally specified using a programming language. Using informal mathematical notation, the initial behavior of `stack` might be written

$$\begin{aligned} b_{[\,]}\text{:}\quad & [c\ \texttt{push}\ x] \mapsto \langle b_{[x\ [\,]]}, \langle\langle c, \texttt{pushed}\rangle\rangle\rangle \\ & [c\ \texttt{pop}] \mapsto \langle b_{[\,]}, \langle\langle c, \texttt{error -- stack empty}\rangle\rangle\rangle \\ & [c\ \texttt{top}] \mapsto \langle b_{[\,]}, \langle\langle c, \texttt{error -- stack empty}\rangle\rangle\rangle \\ & [c\ \texttt{empty?}] \mapsto \langle b_{[\,]}, \langle\langle c, \mathit{true}\rangle\rangle\rangle \end{aligned}$$

where $b_{[x\ y]}$ is the behavior defined by

$$\begin{aligned} b_{[x\ y]}\text{:}\quad & [c\ \texttt{push}\ z] \mapsto \langle b_{[z\ [x\ y]]}, \langle\langle c, \texttt{pushed}\rangle\rangle\rangle \\ & [c\ \texttt{pop}] \mapsto \langle b_y, \langle\langle c, \texttt{popped}\rangle\rangle\rangle \\ & [c\ \texttt{top}] \mapsto \langle b_{[x\ y]}, \langle\langle c, x\rangle\rangle\rangle \\ & [c\ \texttt{empty?}] \mapsto \langle b_{[x\ y]}, \langle\langle c, \mathit{false}\rangle\rangle\rangle \end{aligned}$$

(The mathematical notation here is less precise than the programming language since it does not indicate the values of the behaviors on messages that do not match the patterns.) As a simpler example, the Atolia script

```
accept [ ] dummy
```

signifies the constantly passive behavior

$$p\text{:}\ m \mapsto \langle p, \langle\,\rangle\rangle.$$

It is the purpose of a programming language semantics to define a mapping from syntactic objects such as the code for `stack` to mathematical objects such as the behavior $b_{[\,]}$. The goal of this

[5]That this equation has a solution is assured by the standard theory of programming language semantics. See Dana Scott, "Data types as lattices", *SIAM J Computing* 5, 3, September 1976, pages 522–587, or the books by Stoy or Milne and Strachey.

chapter is to define a mapping from computer programs written in an actor-based language to sets of actor event diagrams representing possible outcomes. The mapping is defined in two stages. In the first stage, the standard denotational theory of sequental programming language semantics is used to define for each program Q in the language a function

$$\mathcal{P}(Q)\colon \mathbf{A} \rightarrow \mathbf{F}$$

giving the initial behavior of each actor. In the second stage that function is used to define the set of possible outcomes of the program Q.

The second stage is largely independent of the programming language. For the purposes of the second stage an actor-based programming language is simply a pair

$$\langle \mathcal{L}, \mathcal{P} \rangle$$

where $\mathcal{L}$ is a description language (set of programs) and $\mathcal{P}$ is a map

$$\mathcal{P}\colon \mathcal{L} \rightarrow (\mathbf{A} \rightarrow \mathbf{F}).$$

Appendix IV presents $\mathcal{L}$ and $\mathcal{P}$ for a toy language illustrating actors.[6] $\mathcal{L}$ and $\mathcal{P}$ have previously been specified for a version of the Act1 language.[7]

IV.3. The Actor Event Diagram Domain

An element of the actor event diagram domain is an actor event diagram as in Chapter II augmented by a (possibly empty) set of pending events. See Figure 2. As before, each vertical line represents an arrival ordering, with time flowing downward so that early events lie above later events. As before, the arrows represent links of the activation ordering. As before, the target and message of an event are written beside the event's dot.

[6]In Appendix IV $\mathcal{L}$ is the set of actor script declarations `Act`. The behavior domain F given in Appendix IV is complicated by actor creation. For programs that do not create actors, the differences between the behavior domain of Appendix IV and the behavior domain of this section may be ignored.

[7]Carl Hewitt and Giuseppe Attardi, unpublishable.

pending:

sieve ← [$filter_2$ reply 3] activated by: $\langle filter_2, 2\rangle$

integers ← [$filter_2$ request] activated by: $\langle filter_2, 2\rangle$

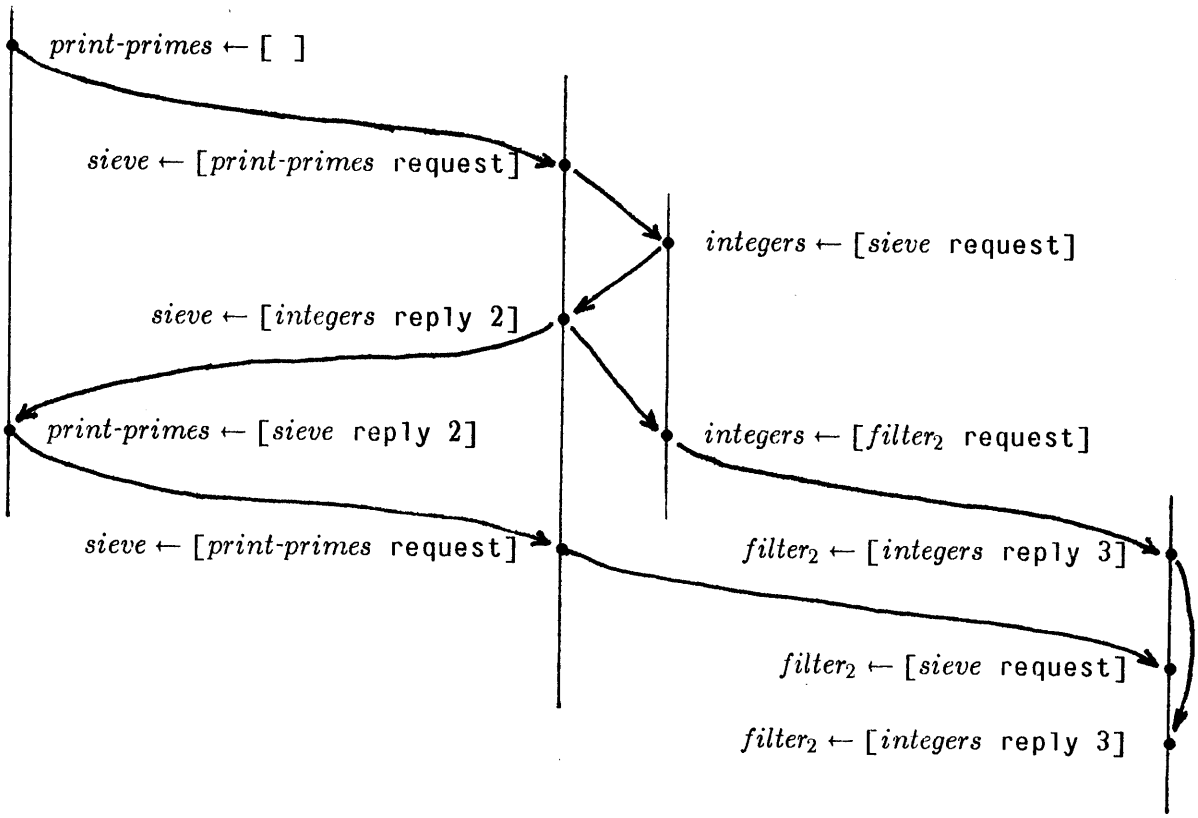


Figure 2. An actor event diagram with pending events.

Each pending event represents a message on its way to a target. The activator of a pending event is the event that caused the message to be sent. When the pending event becomes realized (in a greater element of D) its activator will be the activator of the realized event. In order to refer to arbitrary events, let $\langle a, n\rangle$ be the $(n + 1)$th event in the arrival ordering of a if such exists. In other words, if a is an actor then the successive events in the arrival ordering of a are

$$\langle a, 0\rangle, \langle a, 1\rangle, \langle a, 2\rangle, \langle a, 3\rangle, \langle a, 4\rangle, \ldots.$$

The elements of the actor event diagram domain D will be required to have initial events and to

obey a law of finite activation. Finite activation corresponds to the restriction on primitive serializers that they send out only finitely many messages before they unlock. Simplicity is the only reason for requiring initial events.[8]

The formal definition of the actor event diagram domain will use the concept of *multisets.* A multiset is a set with repetitions. For example, $\{1\}$ and $\{1, 1\}$ are distinct multisets. A *set* with elements from a universe U may be considered a function: $U \to 2$. In like manner a *multiset* with (finite repetitions of) elements from U may be considered a function: $U \to \omega$.[9] The cardinality of a multiset s is defined as

$$\sum_{u \in U} s(u)$$

if the sum exists and is finite, and as ω otherwise since the universe U will always be countable in actor semantics. If s_1 and s_2 are multisets, then their multiset (disjoint) union $s_1 \uplus s_2$ is defined by

$$\forall u \in U \ (s_1 \uplus s_2)(u) = s_1(u) + s_2(u).$$

Similarly their multiset difference $s_1 - s_2$ is defined by

$$\forall u \in U \ (s_1 - s_2)(u) = \begin{cases} 0 & \text{if } s_1(u) \leq s_2(u); \\ s_1(u) - s_2(u) & \text{if } s_1(u) > s_2(u). \end{cases}$$

If s_1: $U_1 \to \omega$ and s_2: $U_2 \to \omega$ are multisets, their multiset product $s_1 \times s_2$: $(U_1 \times U_2) \to \omega$ is defined by

$$\forall \langle u_1, u_2 \rangle \in U_1 \times U_2 \quad (s_1 \times s_2)(\langle u_1, u_2 \rangle) = s_1(u_1) \times s_2(u_2).$$

Let the set of actors, **A**, and the set of messages, **M**, be countable sets.

Definition. *The set of* augmented actor event diagrams *is the set* **D** *of structures*

$$\langle E, M, -act\rightarrow, P \rangle$$

[8]Aside from complicating the discussion of completed elements of the domain, dropping the requirement of an initial event would cause no problems. Finite activation, however, is needed to ensure that the domain has only countably many isolated elements. Extending the essential theorems of Chapter III to domains with uncountably many isolated elements apparently requires the axiom of choice.

[9]This suffices for the semantics given here. Other applications may require a more sophisticated treatment of multisets.

where

- E *is the set of (realized)* events.
- M *is the* message *function.*
- $-act\rightarrow$ *is the* activation ordering.
- P *is the multiset of* pending events.

and the following hold.

- E *is a subset of* $\mathrm{A} \times \omega$ *such that if* $i \leq n$ *and* $\langle a, n\rangle \in E$ *then* $\langle a, i\rangle \in E$.
- M *is a function:* $E \rightarrow \mathbf{M}$.
- $-act\rightarrow$ *is an irreflexive partial order on* E *such that no event has more than one immediate predecessor.*
- P *is a multiset (with finite repetitions) of elements from* $(\mathrm{A} \times \mathrm{M}) \times E$. *That is,* P *is a function:* $((\mathrm{A} \times \mathrm{M}) \times E) \rightarrow \omega$.
- **Finite Delay.** *If* E *is infinite, then* P *is empty.*

Let the target *function* T: $E \rightarrow \mathrm{A}$ *be defined by* $T(\langle a, n\rangle) = a$.

For $a \in \mathrm{A}$, *let the* arrival ordering *of* a, $-arr_a\rightarrow$, *be defined on* E *by*

$$\langle a, i\rangle \; -arr_a\rightarrow \; \langle a', j\rangle \quad \Leftrightarrow \quad a = a' \text{ and } i < j.$$

Let the combined ordering *on* E, $\rightarrow$, *be defined as the transitive closure of*

$$-act\rightarrow \bigcup(\bigcup\{ -arr_a\rightarrow \mid a \in \mathrm{A}\}).$$

- **Law of Strict Causality.** *For no* $e \in E$ *does* $e \rightarrow e$.
- **Law of Countability.** E *is countable.*[10]
- **Law of Finite Predecession.** *For all events* e_1 *the set* $\{ e \mid e \rightarrow e_1 \}$ *is finite.*
- **Initial Event.** *Either* E *and* P *are both empty or there exists an event* e_0 *such that* $\forall e \in E \; e_0 = e$ *or* $e_0 \; -act\rightarrow \; e$.
- **Finite Activation.** *For each* $e \in E$ *the set of events activated by* e *is finite. That is,* $\{ e' \in E \mid e \; -act\rightarrow \; e' \text{ and } \neg\exists e''\; e \; -act\rightarrow \; e'' \; -act\rightarrow \; e' \}$ *is finite.*

[10] This law is redundant here since A is countable and E is a subset of $\mathrm{A} \times \omega$.

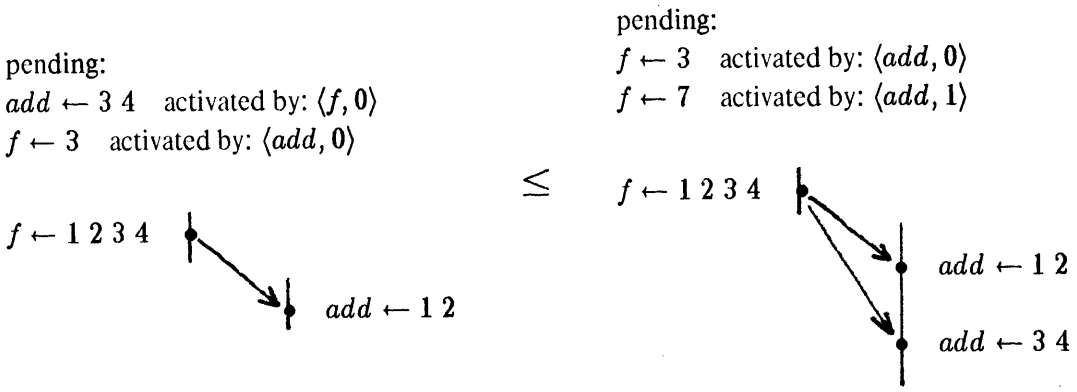


Figure 3. An example of the initial history ordering.

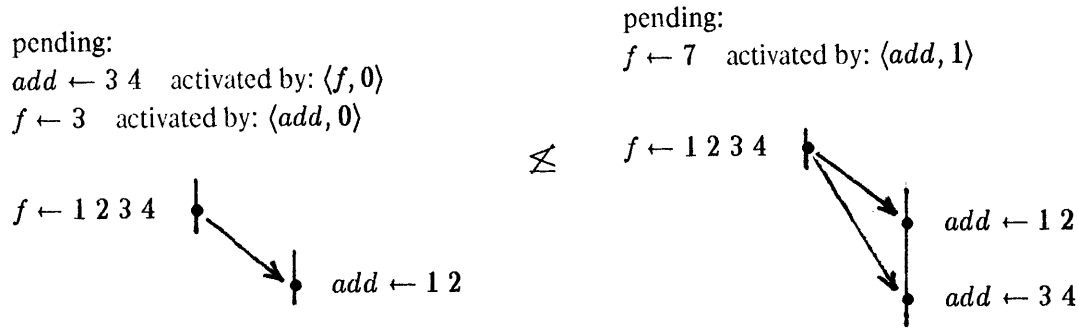


Figure 4. A non-example of the initial history ordering.

• **Finite Activation.** *For each* $e \in E$, *the multiset of pending events activated by* e *is finite. That is,* $\{ \langle \langle a, m \rangle, e \rangle \in P \mid a \in \Lambda, m \in \mathbf{M} \}$ *is a finite multiset.*[11]

(End of definition.)

The partial order to be placed on **D** coincides with the notion of an initial history of a computation. For $x, y \in \mathbf{D}$, $x \leq y$ means that x is a possible stage a computation could go through on its way to y. That is, $x \leq y$ means y could be obtained from x by a process of *expanding* pending events. A

[11] In view of the requirement of Finite Delay, a simpler way to state this is to say that P is a finite multiset. In the completion, however, where Finite Delay does not hold, P can be infinite.

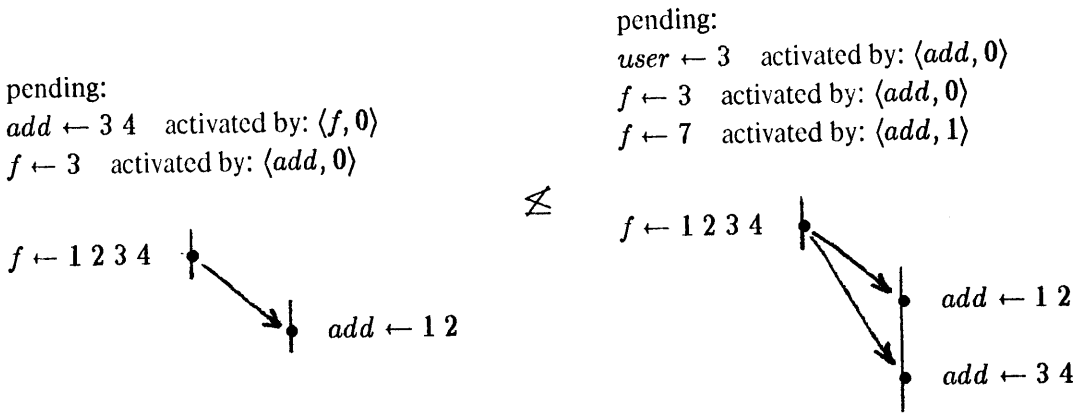


Figure 5. Another non-example of the initial history ordering.

pending event is expanded by making it into an actual (realized) event and adding any pending events that it may activate. Normally the new pending events would be determined by the current behavior of the target of the newly realized event. Since $\leq$ is defined without reference to behaviors, though, $x \leq y$ means that for some assignment of behaviors to actors y can be obtained from x through some sequence of event expansions.

The best way to understand the initial history ordering $\leq$ on **D** is by way of examples and near misses. Figure 3 is an example of $\leq$. Figure 4 is not an example because one of the pending events disappears without being realized. Figure 5 is not an example because a pending event whose activator had already been realized appears out of nowhere.

Definition. *Let* $x = \langle E_x, M_x, \text{—}act\text{→}_x, P_x \rangle \in \mathbf{D}$ *and* $y = \langle E_y, M_y, \text{—}act\text{→}_y, P_y \rangle \in \mathbf{D}$. x *is an* initial history *of* y, *written* $x \leq y$, *if and only if*

- $E_x \subseteq E_y$.
- $\forall e \in E_x \; M_x(e) = M_y(e)$.
- $\forall e, e' \in E_x \quad e \text{—}act\text{→}_x e' \quad \Leftrightarrow \quad e \text{—}act\text{→}_y e'$.
- *Each pending event in* x *is accounted for exactly once in* y, *either as a pending event in* P_y *or as a realized event in* E_y. *Furthermore all the pending events of* y *activated by events already in* x *must be accounted for in this way. More formally, using* $\{\cdot\}$ *to indicate multiset abstraction in which repetitions*

(These diagrams use the notation (e) to indicate the activator e of a pending event. Also the arrival ordering of a is labelled at its top, so only messages are written beside event dots.)

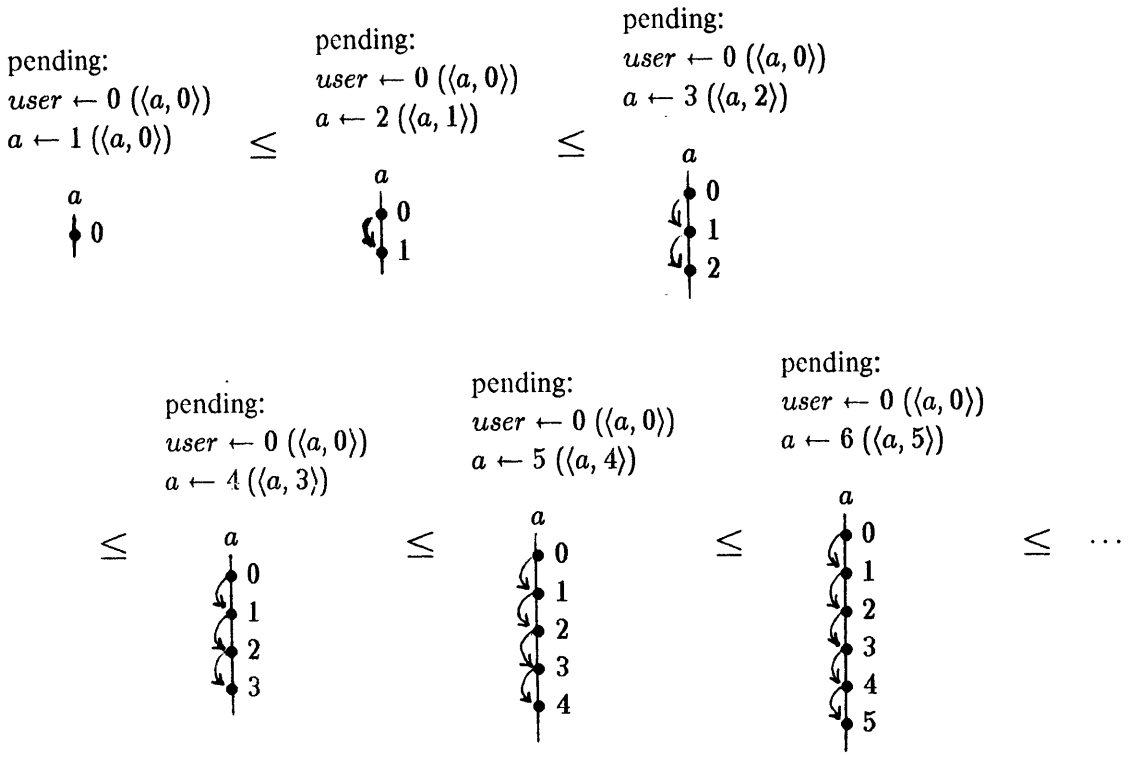


Figure 6. An increasing sequence with no least upper bound.

are counted,

$$P_x = \{\, \langle\langle a, m\rangle, e\rangle \in P_y \mid e \in E_x \,\} \biguplus \{\, \langle\langle T_y(e'), M_y(e')\rangle, e\rangle \mid e' \in E_y - E_x,\ e = \mathit{activator}\,(e'),\ \mathit{and}\ e \in E_x \,\}$$

where $\mathit{activator}\,(e')$ *is the unique immediate predecessor of* e' *in the activation ordering of* y.

Definition. $\langle D, \leq \rangle$ *is the* actor event diagram domain.

The actor event diagram domain is a domain by the definition of Chapter III. The isolated elements are those with a finite number of realized events. The least element has no events at all, realized or pending. The domain is not ω-complete, because there exist increasing sequences having no least upper bound. Figure 6 gives such a sequence in which an event remains pending forever. Though this

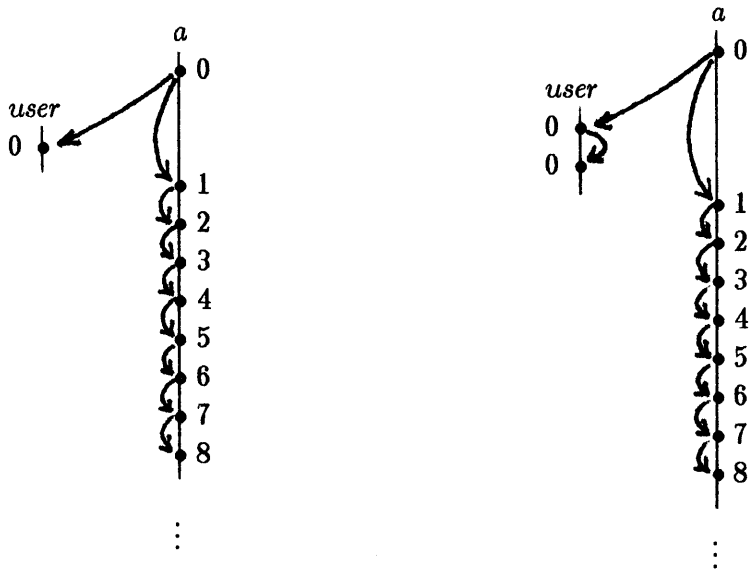


Figure 7. Two upper bounds for the sequence of Figure 6.

sequence has no least upper bound, it has many incomparable minimal upper bounds, and Figure 7 shows two of them.

The least upper bound of a directed set $X \subseteq \mathbf{D}$ will be written $\bigvee X$ if it exists. In view of the following theorem, $\bigvee X$ exists if and only if either (1) the union of the sets of realized events of elements of X is finite; or (2) for every element x of X, for every pending event p of x, there exists $x' \in X$ such that p is realized in x'.

Theorem 1. *If $X \subseteq D$ and $u = \bigvee X$, then for every event e of u there exists $x \in X$ with e an event of x.*

Proof. Suppose u is an upper bound for X in $\mathbf{D}$ and that e is an event of u. If there does not exist $x \in X$ with e an event of x, then it is possible to alter u so as to obtain another upper bound for X incomparable with u. Simply remove from u all activation successors of e and all pending events activated by e or its activation successors, and then rename the remaining realized events. Call the resulting augmented event diagram u'. Since no event following e in the combined ordering of u can be an event of any $x \in X$, u' is also an upper bound for X. Either u and u' are incomparable, or $u = u'$. In the latter case, obtain u'' from $u = u'$ by inserting a new event in the arrival ordering of

$T(e)$ immediately after e, letting e be its activator. u'' is then an upper bound for X incomparable with u. ∎

In the case of Figure 6 any least upper bound would have to have both infinitely many realized events and a pending event. The requirement of Finite Delay thus rules out a least upper bound for the sequence.

The ω-completion of the actor event diagram domain is easily characterized up to isomorphism. Just drop the requirement of Finite Delay from the definition of **D** to obtain its ω-completion $\overline{\mathbf{D}}$.

As noted in §III.5, the event diagrams corresponding to sequential computations have linear activation orderings. In other words, no event activates more than one event. Such event diagrams form an ω-complete subdomain of **D**.

Aside from the least element $\perp$, which represents a computation not yet started, those elements of **D** having no pending events represent computations that have terminated or that have run on to infinity, as distinguished from computations with pending events which represent computations still in progress. Excluding the least element $\perp$, those elements of **D** with no pending events will therefore be called the *completed* elements of **D**. The completed elements may also be characterized as the maximal elements of **D**.

IV.4. Meanings as Fixed Points

Since **D** is a domain, its power domain, $P[\mathbf{D}]$, exists. $P[\mathbf{D}]$ is the semantic domain in which programs written in actor-based languages will be given meanings.

Let Q be a program, with

$$\mathcal{P}(Q)\colon \mathbf{A} \to \mathbf{F}$$

the function giving the initial behavior of each actor as determined by the program Q. These behaviors will be used to define a continuous function f_* on $P[\mathbf{D}]$ whose least fixed point will serve as the meaning of the program Q.

For x an ordered pair, let $x \downarrow 1$ be the first and $x \downarrow 2$ be the second element of the pair. Let

$$next\colon (\mathrm{F} \times (\mathrm{A} \times \mathrm{M})^*) \to \mathrm{F}$$

and

$$pend\colon (\mathrm{F} \times (\mathrm{A} \times \mathrm{M})^*) \to ((\mathrm{A} \times \mathrm{M}) \to \omega)$$

be defined as follows. $next\,(x) = x \downarrow 1$ is the behavior part of the pair. $pend\,(x)$ is the multiset with elements from $\mathrm{A} \times \mathrm{M}$ such that $pend\,(x)\,(\langle a, m\rangle)$ is the number of times that $\langle a, m\rangle$ appears in the sequence $x \downarrow 2$. If ϕ is the behavior of an actor when it accepts the message m of an event e, then $next\,(\phi\, m)$ is the next behavior of the actor, and $pend\,(\phi\, m) \times \{\, e \,\}$ is the multiset of pending events activated by e.

$f_*\colon P[\mathrm{D}] \to P[\mathrm{D}]$ will be defined pointwise from a function $f\colon \mathrm{D} \to P[\mathrm{D}]$, which will in turn be defined from a function $g\colon \mathrm{D} \to P[\mathrm{D}]$. For $x \in \mathrm{D}$, $g\,(x)$ is essentially the set of augmented event diagrams that result from expanding exactly one pending event of x in accord with the actor behaviors specified by the program Q. In fact $g\,(x)$ is a little more, because $g\,(x)$ has to satisfy the closure requirements that hold for elements of the power domain.

The first step in defining g is to define $g\,(\perp)$, which amounts to deciding how program execution should be initiated, which in turn amounts to deciding on an initial event. It is an arbitrary decision, but suppose that execution begins when a special message m_0 arrives at a particular actor a_0 singled out by the language. (For the toy language described in the appendixes, m_0 is the empty sequence $\langle\,\rangle$ and the target of the initial event is $\langle program, 0\rangle$.)

Therefore let

$$g\,(\perp) = \{\, \langle E, M, -act\rightarrow, P\rangle \,\}^c$$

where ^{c} is the closure operation defined in §III.3 and

$$\begin{aligned} E &= \{\, \langle a_0, 0\rangle \,\} \\ M(\langle a_0, 0\rangle) &= m_0 \\ -act\rightarrow &= \emptyset \\ P &= pend\,(\mathfrak{P}\,(Q)\, a_0\, m_0) \times \{\, \langle a_0, 0\rangle \,\}. \end{aligned}$$

Now to define g on $x = \langle E, M, -act\rightarrow, P\rangle \in \mathbf{D}$, where $x \neq \perp$. Let *behavior* (a) be the current behavior of actor a in x, that is, the behavior of actor a after it has accepted the messages of all the events in its arrival ordering. More formally, define the successive behaviors of a by

$$b(a, 0) = \mathcal{P}(Q)\, a$$
$$b(a, n+1) = next\,(b\,(a, n)\,(M(\langle a, n\rangle)))$$

and let *behavior* $(a) = b\,(a, n)$ where n is the least integer such that $\langle a, n\rangle \notin E$. If there are infinitely many events in the arrival ordering of a, so that no such n exists, then the current behavior of a is undefined.

If P is empty, there are no pending events to expand and so let

$$g(x) = \{x\}^c$$

where ^{c} is the closure operation defined in §III.3. Otherwise for each $p = \langle\langle a, m\rangle, e\rangle \in P$, or more properly for each $p \in ((\mathrm{A} \times \mathrm{M}) \times E)$ such that $P(p) \neq 0$, let $x\,(a, m, e)$ be the element of $\mathbf{D}$ obtained by (1) adding a new event to x with target a, message m, and activator e; (2) subtracting the pending event p from the pending events of x; and (3) adding the pending events activated by the new event. Then define

$$g(x) = \{\, x\,(a, m, e) \mid \langle\langle a, m\rangle, e\rangle \in P \,\}^c.$$

To define $x\,(a, m, e)$ more precisely, let n be the least integer such that $\langle a, n\rangle \notin E$. Such an n must exist because the existence of pending events implies that the set of events is finite. Then

$$x\,(a, m, e) = \langle E', M', -act\rightarrow', P'\rangle$$

where

$$E' = E \bigcup \{\langle a, n\rangle\}$$

$$M'(e) = \begin{cases} M(e) & \text{if } e \in E \\ m & \text{if } e = \langle a, n\rangle \end{cases}$$

$$e_1 -act\rightarrow' e_2 \Leftrightarrow (e_1, e_2 \in E \text{ and } e_1 -act\rightarrow e_2)$$
$$\text{or } (e_2 = \langle a, n\rangle \text{ and } (e_1 = e \text{ or } e_1 -act\rightarrow e))$$

$$P' = (P - \{\langle\langle a, m\rangle, e\rangle\}) \biguplus (pend\,(behavior\,(a)\,m) \times \{\langle a, n\rangle\})$$

This completes the definition of g: $\mathbf{D} \to P[\mathbf{D}]$. Define f: $\mathbf{D} \to P[\mathbf{D}]$ by

$$f(x) = \bigsqcup_{z \leq x} g(z).$$

Some observations:

- If $\langle\langle a, m\rangle, e\rangle$ is a pending event of x, then $x \leq x(a, m, e)$.
- If $x \leq y$ then $x \in g(y)$. In particular $x \in f(x)$.
- If x is isolated, then $g(x)$ contains only isolated elements. This follows from the property of $\mathbf{D}$ that if $y \in \mathbf{D}$ is isolated, and $x \leq y$, then x is also isolated.
- If x is not isolated, then $g(x)$ contains exactly one element that is not isolated, namely x. This follows from the fact that elements of $\mathbf{D}$ that are not isolated are maximal in $\mathbf{D}$.

Theorem 1. *f: $\mathbf{D} \to P[\mathbf{D}]$ is ω-continuous.*

Proof. Let $x \leq y$. Then

$$f(x) = \bigsqcup_{z \leq x} g(z) \sqsubseteq \bigsqcup_{z \leq y} g(z) = f(y)$$

so f is monotonic.

To show f ω-continuous, let $\{x_i\}_{i \in \omega}$ be an increasing sequence in $\mathbf{D}$ having a least upper bound $x = \bigvee_{i \in \omega} x_i$. Let $z \leq x$. If z is isolated, then there exists x_i with $z \leq x_i$, whence

$$g(z) \sqsubseteq \bigsqcup_{z \leq x_i} g(z) = f(x_i) \sqsubseteq \bigsqcup_{i \in \omega} f(x_i).$$

On the other hand, if z is not isolated then $z = x$ and $g(z) = \{x\}^c$. $x_i \in f(x_i)$ for each i, so

$$x \in (\bigcup_{i \in \omega} f(x_i))^c = \bigsqcup_{i \in \omega} f(x_i).$$

Hence

$$g(z) = \{x\}^c \sqsubseteq \bigsqcup_{i \in \omega} f(x_i).$$

Thus

$$\bigcup_{z \leq x} g(z) \subseteq \bigsqcup_{i \in \omega} f(x_i)$$

and applying the closure operation to both sides yields

$$f(x) \sqsubseteq \bigsqcup_{i \in \omega} f(x_i).$$

Monotonicity implies the reverse direction, so f is ω-continuous. ∎

Define f_*: $P[\mathbf{D}] \to P[\mathbf{D}]$ by

$$f_*(A) = \bigsqcup\{ f(x) \mid x \in A \} = (\bigcup_{x \in A} f(x))^c.$$

$f_*(A)$ thus consists of the augmented event diagrams in A together with all the event diagrams that can be had by taking an element of A and expanding one of its pending events.

The following general theorem shows that f_* is ω-continuous.

Theorem 2. *Let D be a domain, and let f: $D \to P[D]$ be ω-continuous. Then f_*: $P[D] \to P[D]$ defined by*

$$f_*(A) = \bigsqcup_{x \in A} f(x)$$

is ω-continuous.

Proof. Monotonicity is obvious. Hence it suffices to show

$$f_*(\bigsqcup_{i \in \omega} A_i) \sqsubseteq \bigsqcup_{i \in \omega} f_*(A_i)$$

for increasing sequences

$$A_0 \sqsubseteq A_1 \sqsubseteq A_2 \sqsubseteq A_3 \sqsubseteq \cdots.$$

Let

$$c \in f_*(\bigsqcup_{i \in \omega} A_i) = \bigsqcup\{ f(x) \mid x \in \bigsqcup_{i \in \omega} A_i \}$$

$$= (\bigcup\{ f(x) \mid x \in (\bigcup_{i \in \omega} A_i)^c \})^c.$$

Let $W \subseteq \bigcup\{f(x) \mid x \in (\bigcup_{i\in\omega} A_i)^c\}$ be a directed set with $c = \bigvee W$. For $w \in W$ let $Z_w \subseteq \bigcup_{i\in\omega} A_i$ be a directed set with

$$w \in f(\bigvee Z_w) = \bigsqcup_{z\in Z_w} f(z) = (\bigcup_{z\in Z_w} f(z))^c.$$

Let $Y_w \subseteq \bigcup_{z\in Z_w} f(z)$ be a directed set of isolated elements with $w = \bigvee Y_w$. Let $Y = \bigcup_{w\in W} Y_w$. Then

$$c = \bigvee W = \bigvee(\bigcup_{w\in W} Y_w) = \bigvee Y$$

and

$$\begin{aligned} Y &= \bigcup_{w\in W} Y_w \\ &\subseteq \bigcup_{w\in W} \bigcup_{z\in Z_w} f(z) \\ &\subseteq \bigcup_{i\in\omega} \bigcup_{z\in A_i} f(z) \\ &\subseteq \bigcup_{i\in\omega} f_*(A_i). \end{aligned}$$

Furthermore, Y is directed: if $y_1 \in Y_{w_1}$ and $y_2 \in Y_{w_2}$, then let w_3 be an upper bound for w_1 and w_2 in W. Since y_1 and y_2 are isolated, Y_{w_3} is directed, and

$$y_1, y_2 \leq w_3 = \bigvee Y_{w_3},$$

there exists $y_1', y_2' \in Y_{w_3}$ with $y_1 \leq y_1'$ and $y_2 \leq y_2'$. Let y_3 be an upper bound for y_1' and y_2' in Y_{w_3}. y_3 is then an upper bound for y_1 and y_2 in Y.

Therefore

$$c \in (\bigcup_{i\in\omega} f_*(A_i))^c = \bigsqcup_{i\in\omega} f_*(A_i).$$

∎

Being continuous, f_*: $P[\mathbf{D}] \to P[\mathbf{D}]$ has a least fixed point

$$A_Q = \bigsqcup_{i\in\omega} f_*^i(\bot).$$

Define the meaning of the program Q to be that least fixed point A_Q. The theorems below show that A_Q is the set of initial histories of the actor event diagrams that correspond to completed computations of Q.

They also show that this power domain semantics is compatible with Greif's behavioral semantics.[12] Behavioral semantics does not use pending events or fixed points. Instead it uses *causal axioms* to go directly from behaviors to the set of completed computations. Essentially these causal axioms state that a completed computation is an actor event diagram that is complete with respect to the initial behaviors.

Definition. *An augmented actor event diagram* $x = \langle E, M, \text{—}act\text{→}, P\rangle \in \mathbf{D}$ *is* consistent with respect to *the initial behaviors given by*

$$\mathcal{P}(Q)\colon \mathbf{A} \to \mathbf{F}$$

iff for each event $e = \langle a, n\rangle \in E$

$$pend\,(b\,(a, n)\,(M(e))) = \{\,\langle a', m'\rangle \mid \langle\langle a', m'\rangle, e\rangle \in P\,\} \biguplus \{\,\langle T(e'), M(e')\rangle \mid e' \in E \text{ and } e = activator\,(e')\,\}$$

(where $\{\cdot\}$ *indicates multiset abstraction in which repetitions are counted). In other words, the pending and realized events activated by* e *are as they should be according to the behavior of* $a = T(e)$ *at the time of the event* e.

Definition. *An augmented actor event diagram* $x = \langle E, M, \text{—}act\text{→}, P\rangle \in \mathbf{D}$ *is* complete with respect to *the initial behaviors given by*

$$\mathcal{P}(Q)\colon \mathbf{A} \to \mathbf{F}$$

iff $x \neq \perp$, P *is empty, and for each event* $e = \langle a, n\rangle \in E$

$$pend\,(b\,(a, n)\,(M(e))) = \{\,\langle T(e'), M(e')\rangle \mid e' \in E \text{ and } e = activator\,(e')\,\}$$

(where $\{\cdot\}$ *indicates multiset abstraction in which repetitions are counted). In other words,* x *has at least one event,* x *has no pending events, and* x *is consistent with respect to the initial behaviors given by* $\mathcal{P}(Q)$.

[12] Irene Greif, "Semantics of communicating parallel processes", MIT Project MAC Technical Report 154, September 1975.

The following theorems prove that the actor event diagrams that are complete with respect to the initial behaviors given by $\mathcal{P}(Q)$ are precisely the completed elements of the least fixed point. (Recall that the completed elements of **D** are just the maximal elements of **D.**)

Theorem 3. *Every element of the least fixed point A_Q is consistent with respect to the initial behaviors given by $\mathcal{P}(Q)$.*

Proof. Refer to the definition of the initial history ordering $\leq$ and the theorem following it in §3. If $x \leq y$, and y is consistent with respect to the initial behaviors, then x is also. If $X \subseteq \mathbf{D}$ is directed, each element of X is consistent with respect to the initial behaviors, and $\bigvee X$ exists, then $\bigvee X$ is consistent with respect to the initial behaviors. Thus if every element of $Y \subseteq \mathbf{D}$ is consistent with respect to the initial behaviors, then so is every element of Y^c. It follows that if for $\alpha \in I$ each element of $A_\alpha \in P[\mathbf{D}]$ is consistent, then each element of $\bigsqcup_{\alpha \in I} A_\alpha$ is consistent.

Hence most of the operations involved in the construction of $A_Q = \bigsqcup_{i \in \omega} f_*^i(\perp)$ preserve consistency. $\perp$ is consistent with respect to any initial behaviors. There remains only to show that if $x \in \mathbf{D}$ is consistent, then the elements of $g(x)$ are consistent.

Both elements of $g(\perp)$ are consistent. If x is consistent and has no pending events, then $g(x) = \{x\}^c$ so the elements of $g(x)$ are consistent. If x is consistent and has $\langle\langle a, m\rangle, e\rangle$ as a pending event, then $x(a, m, e)$ is consistent. Thus $g(x)$ contains only consistent elements.

Therefore every element of A_Q is consistent with the initial behaviors given by $\mathcal{P}(Q)$. ∎

Theorem 4.

$$A_Q = \{ x \in \mathbf{D} \mid x \text{ is consistent with respect to the initial behaviors given by } \mathcal{P}(Q) \}.$$

Proof. The preceding theorem takes care of the forward inclusion.

Let $x = \langle E, M, \text{—}act\text{→}, P\rangle \in \mathbf{D}$ be consistent with respect to the initial behaviors given by $\mathcal{P}(Q)$. By Theorem 1 of §II.5 there exists a one-to-one mapping g: $E \to \omega$ that preserves the combined ordering $\to$. For $i \in \omega$, let x_i be the unique element of **D** such that $x_i \leq x$ and x_i has

$$\{ e \in E \mid g(e) = j \text{ for some } j < i \}$$

as its set of realized events. Then for each $i \in \omega$

$$x_i \in f_*^i(\perp)$$

and so $x = \bigvee_{i\in\omega} x_i \in \bigsqcup_{i\in\omega} f_*^i(\perp)$. ∎

Definition. *If* $A \in P[\mathbf{D}]$, *then*

$$\text{completed}(A) = \{\, x \in A \mid x \textit{ is maximal in } \mathbf{D} \,\}.$$

Corollary 5.

$$\text{completed}(A_Q) = \{\, x \in \mathbf{D} \mid x \textit{ is complete with respect to the initial behaviors given by } \mathcal{P}(Q) \,\}.$$

§8 describes a power domain isomorphic to $\langle P[\mathbf{D}], \sqsubseteq\rangle$ in which the least fixed point of f_* contains only the completed elements.

The following theorem confirms a claim made in §III.5.

Theorem 6. *Every element of the least fixed point* A_Q *is an initial history of a completed element of* A_Q.

Proof. Let $x \in A_Q = \bigsqcup_{i\in\omega} f_*^i(\perp)$. Either x is itself completed or $x \in f_*^n(\perp)$ for some $n \in \omega$. In the first case there is nothing to prove.

In the second case it is possible to construct an increasing sequence in A_Q beginning with x that has a completed least upper bound. Let the pending events of x be $p_0 = \langle\langle a_0, m_0\rangle, e_0\rangle, \ldots, p_{k_0}$. Let $x_0 = x$.

Let $x_1 = x_0(a_0, m_0, e_0)$ and let $p_1, \ldots, p_{k_0}, \ldots, p_{k_1}$ be the pending events of x_1, where $p_1, \ldots, p_{k_0}$ are the same as before.

The induction hypothesis for i is that for all $j < i$ $x_j \leq x_{j+1}$, and for all $j \leq i$ $x_j \in f_*^{n+j}(\perp)$ and either x_j is completed or the pending events of x_j are $p_j, \ldots, p_{k_j}$. If x_i is completed,

define $x_{i+1} = x_i$. Otherwise define $x_{i+1} = x_i(a_i, m_i, e_i)$ where $p_i = \langle\langle a_i, m_i\rangle, e_i\rangle$. If x_{i+1} is not completed then let the pending events of x_{i+1} be $p_{i+1}, \ldots, p_{k_i}, \ldots, p_{k_{i+1}}$ where $p_{i+1}, \ldots, p_{k_i}$ are the same as before.

The least upper bound of the sequence $\{ x_i \}_{i \in \omega}$ exists and is a completed element of A_Q. ∎

What has been accomplished?

Augmented by pending events, the actor event diagrams form a domain under the initial history ordering. Although the actor event diagram domain is incomplete, its power domain exists and provides a fixed point semantics for actor-based languages. This power domain semantics, which is denotational, is compatible with behavioral semantics.

The actor power domain shows that a power domain whose underlying domain is incomplete can deal with finite delay and the unbounded nondeterminism that results.

IV.5. Example: Infinite Loop

This section calculates the fixed point for a program that loops forever. It is interesting to compare this example with that of the next section.

As in the examples of the next two sections, there are only two actors to consider. One of the actors is the *user*, which simply accepts messages. The other actor is a_0, the target of the initial event. Its behavior is defined by an Atolia program.[13] In this instance, the program is

```
(loop = accept [ ]
        send "add1" to loop ;
        become i initially 0
                 inside
               accept [ msg ]
               if equal [ msg "add1" ]
                  then change i to plus [ i 1 ] ;
                       send "add1" to loop
                  else
               if equal [ msg "halt" ]
                  then send i to user ; become accept [ ] dummy
                  else dummy).
```

[13] See §V.5.

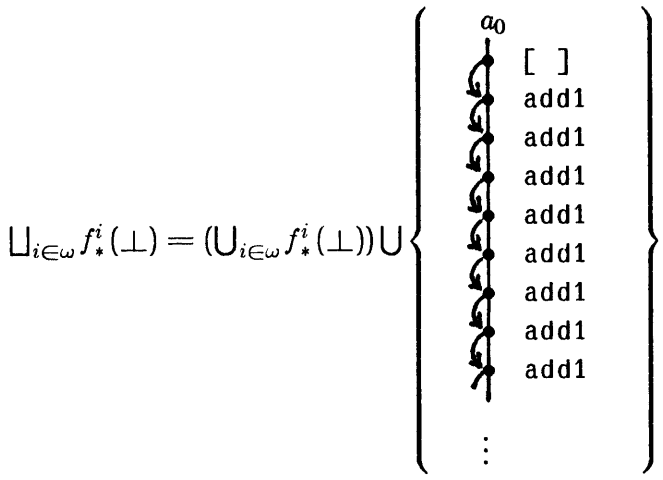


Figure 9. The least fixed point $\bigsqcup_{i\in\omega} f_*^i(\perp)$ for `loop`.

This program says that when a_0 receives the go message m_0 (which will be written [ ] in event diagrams) it initializes itself to a state 0 and sends itself an increment instruction. When it accepts an increment instruction in state i, it enters state $i + 1$ and sends itself another increment instruction. Were it ever to accept a halt instruction, it would tell the *user* its current state. Its initial behavior is b: M $\rightarrow$ (F $\times$ (A $\times$ M)* given by

$$\begin{aligned}
b\text{:} \quad & m_0 \mapsto \langle b_0, \langle [a_0 \leftarrow \texttt{add1}] \rangle \rangle \\
b_i\ (i \in \omega)\text{:} \quad & \texttt{add1} \mapsto \langle b_{i+1}, \langle [a_0 \leftarrow \texttt{add1}] \rangle \rangle \\
& \texttt{halt} \mapsto \langle \mathit{passive}, \langle [\mathit{user} \leftarrow i] \rangle \rangle \\
\mathit{passive}\text{:} \quad & m \mapsto \langle \mathit{passive}, \langle\ \rangle \rangle
\end{aligned}$$

where $[t \leftarrow m]$ indicates the ordered pair $\langle t, m \rangle$ signifying that the message m is sent to the target t. Messages that do not match one of the cases given are just ignored.

It is easy to calculate the least upper bound of the function f_*: P[D] $\rightarrow$ P[D] associated with this very simple program. The stages $f_*^i(\perp)$ are shown in Figure 8. The least fixed point is shown in Figure 9, and the lone completed element of the least fixed point is shown in Figure 10.

The event diagrams in these figures are drawn compactly, with each actor's arrival ordering

$$f_*^0(\perp) = \{\perp\}$$

$$f_*^1(\perp) = f_*^0(\perp) \cup \left\{ \begin{array}{l} \text{pending:} \\ a_0 \leftarrow \texttt{add1}\ (\langle a_0, 0\rangle) \\ \\ \quad a_0 \\ \quad \bullet\ [\ ] \end{array} \right\}$$

$$f_*^2(\perp) = f_*^1(\perp) \cup \left\{ \begin{array}{l} \text{pending:} \\ a_0 \leftarrow \texttt{add1}\ (\langle a_0, 1\rangle) \\ \\ \quad a_0 \\ \quad \bullet\ [\ ] \\ \quad \bullet\ \texttt{add1} \end{array} \right\}$$

$$f_*^3(\perp) = f_*^2(\perp) \cup \left\{ \begin{array}{l} \text{pending:} \\ a_0 \leftarrow \texttt{add1}\ (\langle a_0, 2\rangle) \\ \\ \quad a_0 \\ \quad \bullet\ [\ ] \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \end{array} \right\}$$

$$f_*^4(\perp) = f_*^3(\perp) \cup \left\{ \begin{array}{l} \text{pending:} \\ a_0 \leftarrow \texttt{add1}\ (\langle a_0, 3\rangle) \\ \\ \quad a_0 \\ \quad \bullet\ [\ ] \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \end{array} \right\}$$

$$f_*^5(\perp) = f_*^4(\perp) \cup \left\{ \begin{array}{l} \text{pending:} \\ a_0 \leftarrow \texttt{add1}\ (\langle a_0, 4\rangle) \\ \\ \quad a_0 \\ \quad \bullet\ [\ ] \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \\ \quad \bullet\ \texttt{add1} \end{array} \right\}$$

and so on.

Figure 8. $f_*^i(\perp)$ for `loop`.

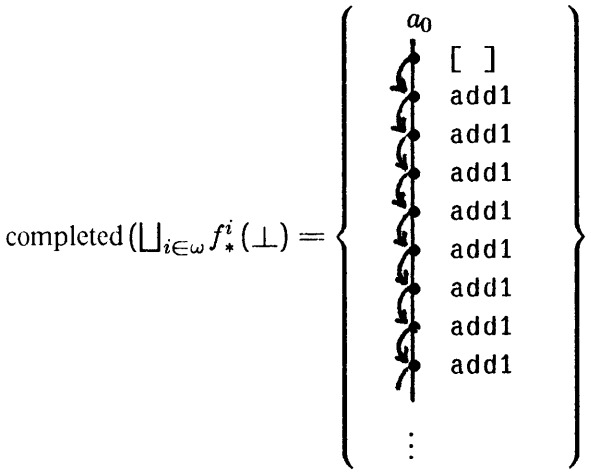


Figure 10. completed $(\bigsqcup_{i\in\omega} f_*^i(\perp))$ for `loop`.

labelled at its top by the name of the actor so only messages need be written beside events. The activator of a pending event appears in parentheses after the pending event. Recall that $\langle a, n\rangle$ is the (n+1)th event in the arrival ordering of a.

For all programs the stages $f_*^i(\perp)$ contain finite initial histories only. It is the least upper bound operation $\bigsqcup$ that puts elements representing nonterminating computations into the least fixed point. The least upper bound of $\{f_*^i(\perp) \mid i \in \omega\}$ consists of the union $\bigcup_{i\in\omega} f_*^i(\perp)$ together with all existing least upper bounds in D of strictly increasing[14] sequences of elements from the union. In this example all strictly increasing sequences of elements from the union have the same least upper bound, so the least fixed point of f_* contains only one event diagram that does not appear in any $f_*^i(\perp)$.

In the next example the strictly increasing sequences of elements from $\bigcup_{i\in\omega} f_*^i(\perp)$ have no least upper bound, so the least fixed point is the same as the union.

IV.6. Example: Terminating Unbounded Choice

The following program has unbounded nondeterminism.

[14] A strictly increasing sequence $\{x_i\}_{i\in\omega}$ has $x_i \leq x_{i+1}$ and $x_i \neq x_{i+1}$ for all $i \in \omega$.

```
(choose = accept [ ]
          send "add1" to choose ;
          send "halt" to choose ;
          become i initially 0
                   inside
                 accept [ msg ]
                 if equal [ msg "add1" ]
                    then change i to plus [ i 1 ] ;
                         send "add1" to choose
                    else
                 if equal [ msg "halt" ]
                    then send i to user ; become accept [ ] dummy
                    else dummy)
```

This program is almost the same as the `loop` program given in the last section. When actor a_0 accepts the go message m_0, it initializes itself to a state 0 and sends itself both an increment instruction and a halt instruction. Since all messages must eventually arrive at their targets, a_0 will eventually accept this halt instruction and terminate. Unlike the `loop` program, then, the `choose` program must terminate.

The initial behavior of a_0 is b given by

$$\begin{aligned}
b\text{:} \quad & m_0 \mapsto \langle b_0, \langle [a_0 \leftarrow \texttt{add1}], [a_0 \leftarrow \texttt{halt}] \rangle \rangle \\
b_i\ (i \in \omega)\text{:} \quad & \texttt{add1} \mapsto \langle b_{i+1}, \langle [a_0 \leftarrow \texttt{add1}] \rangle \rangle \\
& \texttt{halt} \mapsto \langle \mathit{passive}, \langle [\mathit{user} \leftarrow i] \rangle \rangle \\
\mathit{passive}\text{:} \quad & m \mapsto \langle \mathit{passive}, \langle\ \rangle \rangle .
\end{aligned}$$

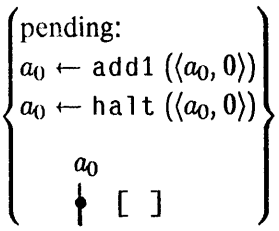
Again it is easy to calculate the least upper bound of the function f_*: $P[\mathbf{D}] \rightarrow P[\mathbf{D}]$. The stages $f_*^i(\perp)$ are shown in Figure 11. The least fixed point is

$$\bigcup_{i \in \omega} f_*^i(\perp),$$

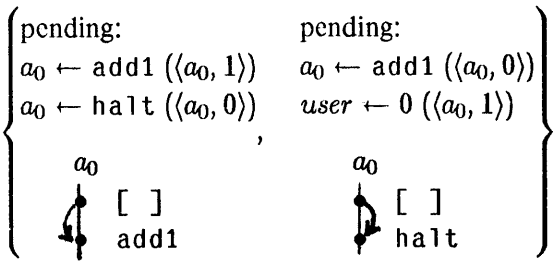
that is, the union of the stages in Figure 11, and the set of completed elements of the least fixed point is shown in Figure 12. There are no elements representing nonterminating computations in the least fixed point because the strictly increasing sequences of elements from $\bigcup_{i \in \omega} f_*^i(\perp)$, such

$f_*^0(\perp) = \{\perp\}$

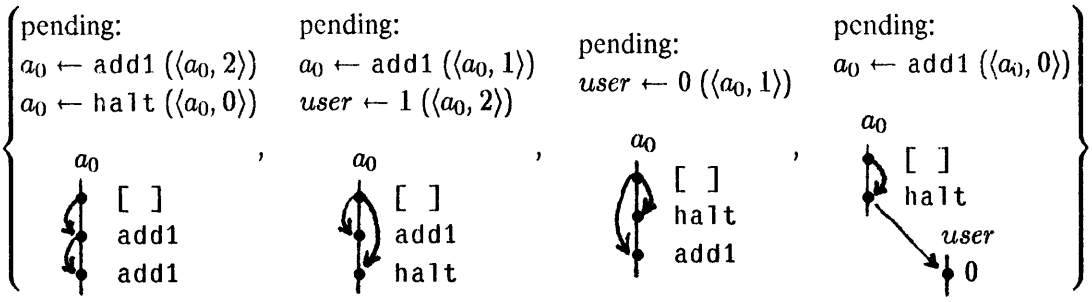
$f_*^1(\perp) = f_*^0(\perp) \cup$



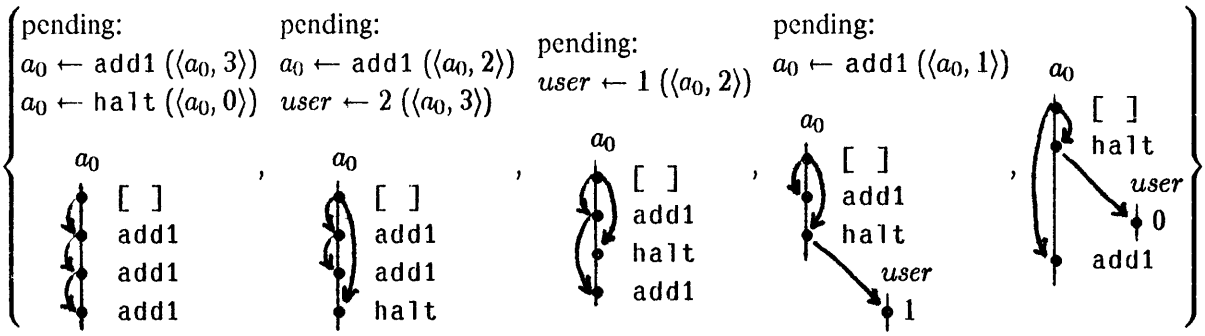
$f_*^2(\perp) = f_*^1(\perp) \cup$



$f_*^3(\perp) = f_*^2(\perp) \cup$



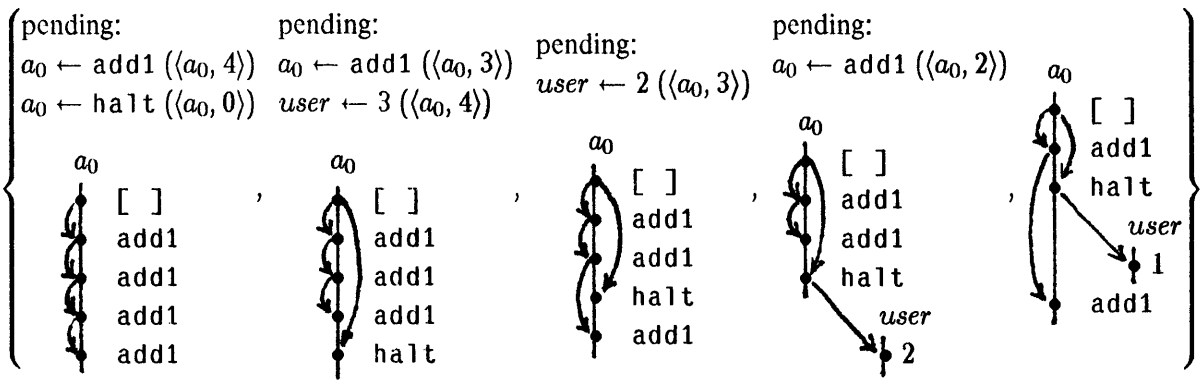
$f_*^4(\perp) = f_*^3(\perp) \cup$



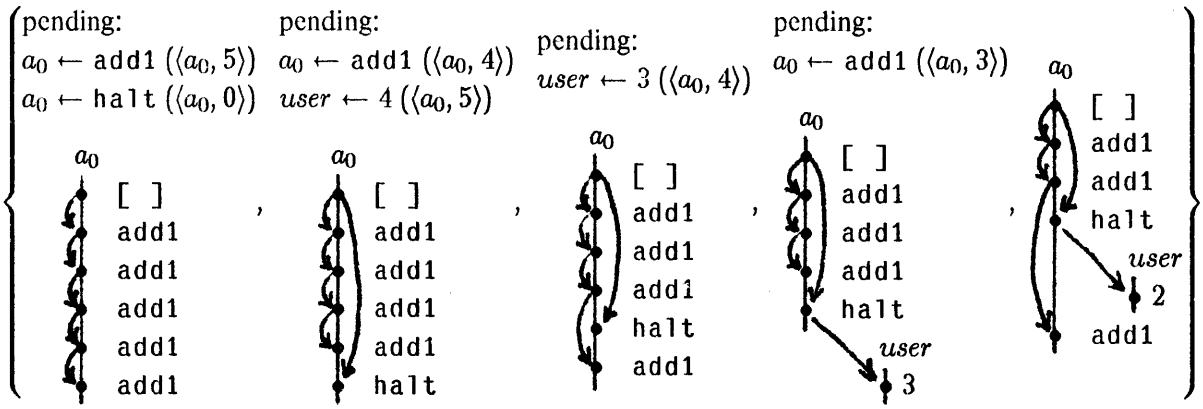
(Continued on next page.)

Figure 11. $f_*^i(\perp)$ for choose.

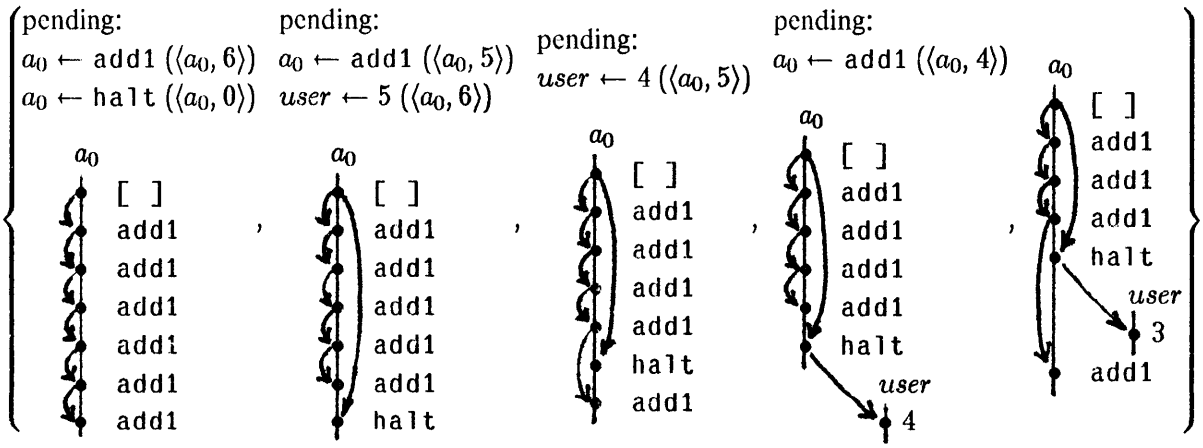
$f_*^5(\perp) = f_*^4(\perp) \cup$



$f_*^6(\perp) = f_*^5(\perp) \cup$



$f_*^7(\perp) = f_*^6(\perp) \cup$



and so on.

Figure 11. (Continued from previous page.) $f_*^i(\perp)$ for **choose**.

completed$(\bigsqcup_{i\in\omega} f_*^i(\bot)) =$

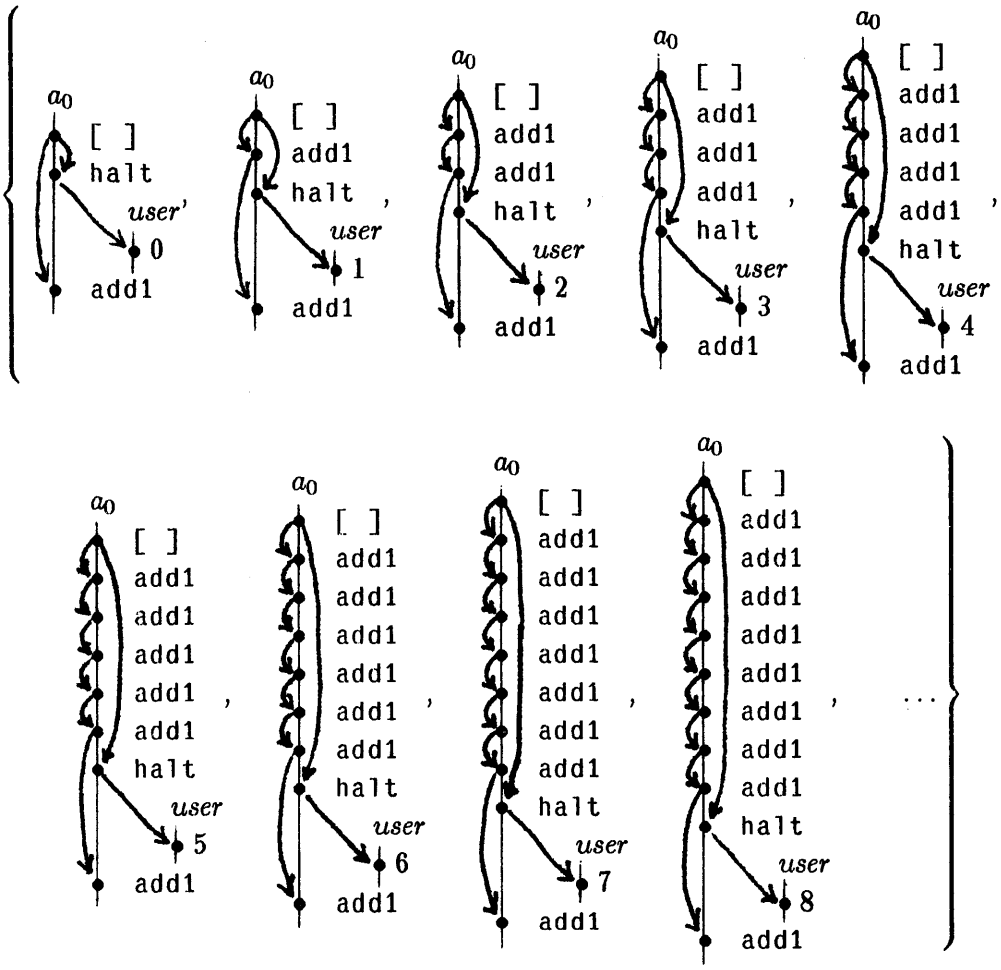


Figure 12. completed$(\bigsqcup_{i\in\omega} f_*^i(\bot))$ for `choose`.

as the sequence in Figure 13, do not have least upper bounds in **D**. Had the power domain been formed from the ω-completion $\overline{\mathbf{D}}$, the least fixed point would contain an element of $\overline{\mathbf{D}}$ representing a nonterminating computation in which an event remains pending forever.

In the existing implementations of Atolia on sequential machines, the `choose` program always produces 0, 1, or 2 when run all by itself. This is allowed by loose nondeterminism: implementations

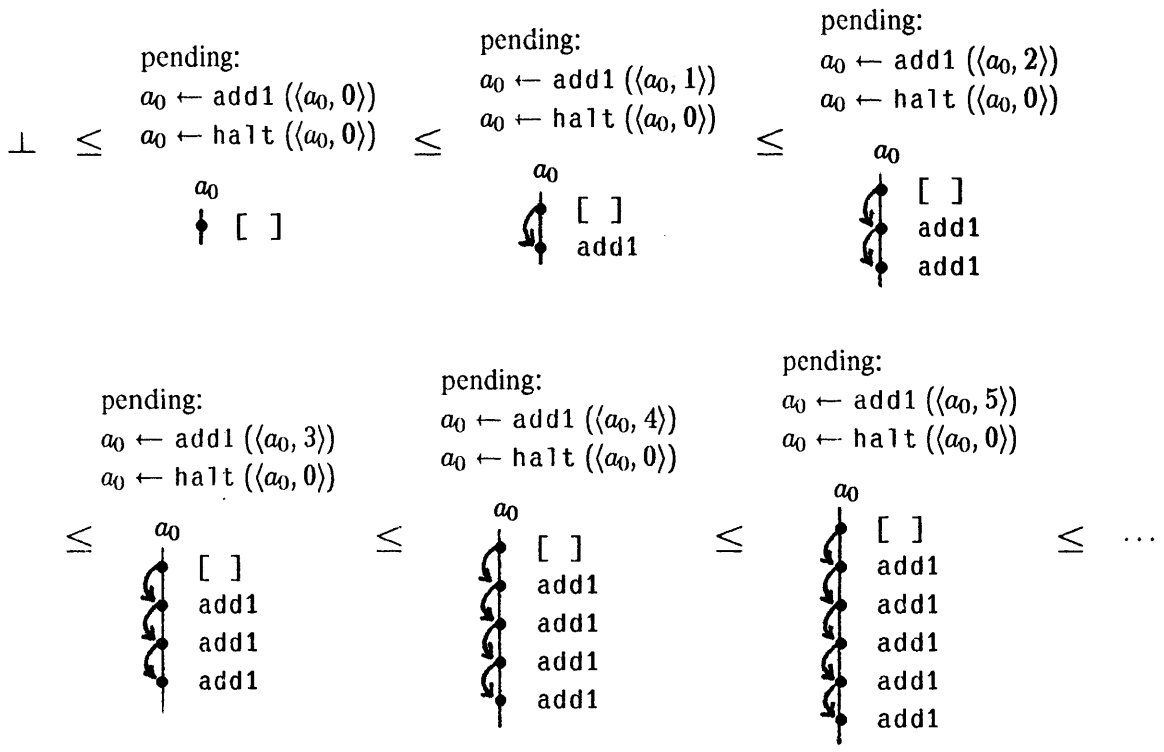


Figure 13. A strictly increasing sequence in $\bigcup_{i \in \omega} f_*^i(\perp)$ with no least upper bound.

are not required to preserve all the nondeterminism present in the semantics.

Even in the existing implementations, however, `choose` can return a result greater than 2 when other programs run pseudo-concurrently. Every bound that might be placed on the result can be exceeded by placing a sufficiently heavy burden on the Atolia processor.

IV.7. Example: Possibly Nonterminating Choice

Sequential programs with choice points are sometimes used in attempts to model nondeterministic concurrency. Such attempts are bound to fail since choice nondeterminism is bounded. This section shows how arrival nondeterminism can successfully model choice nondeterminism.

The program below uses an "arrives-first" choice. Its nondeterminism is bounded.

```
(choice-loop = accept [ ]
               send "add1" to choice-loop ;
               send "stop" to choice-loop ;
               become i initially 0 ;
                      waiting initially false
                        inside
                      accept [ msg ]
                      if waiting
                         then if equal [ msg "stop" ]
                                 then change i to plus [ i 1 ] ;
                                      change waiting to false ;
                                      send "add1" to choice-loop ;
                                      send "stop" to choice-loop
                                 else dummy
                         else if equal [ msg "add1" ]
                                 then change waiting to true
                                 else
                              if equal [ msg "stop" ]
                                 then send i to user ;
                                      become accept [ ] dummy
                                 else dummy)
```

When actor a_0 accepts the go message m_0, it initializes itself to a state 0 and sends itself both an increment instruction and a stop instruction. It obeys whichever instruction arrives first. That is, if a_0 is in state i and the stop instruction arrives first, then a_0 sends i to the user and terminates. If the increment instruction arrives first, though, then a_0 waits until the stop instruction arrives. When the stop instruction arrives, instead of stopping a_0 enters state $i+1$ and begins the cycle again by sending itself both an increment instruction and a stop instruction. The initial behavior of a_0 is b where

$$b: \quad m_0 \mapsto \langle b_0, \langle [a_0 \leftarrow \texttt{add1}], [a_0 \leftarrow \texttt{stop}] \rangle \rangle$$

$$b_i: \quad \texttt{add1} \mapsto \langle wait_i, \langle\,\rangle \rangle$$

$$\texttt{stop} \mapsto \langle passive, \langle [user \leftarrow i] \rangle \rangle$$

$$wait_i\ (i \in \omega): \quad \texttt{stop} \mapsto \langle b_{i+1}, \langle [a_0 \leftarrow \texttt{add1}], [a_0 \leftarrow \texttt{stop}] \rangle \rangle$$

$$passive: \quad m \mapsto \langle passive, \langle\,\rangle \rangle$$

The stages $f_*^i(\perp)$ for the function f_*^i associated with this program are shown in Figure 14. The least fixed point is shown in Figure 15, and the set of completed elements of the least fixed point

$f_*^0(\perp) = \{\perp\}$

$f_*^1(\perp) = f_*^0(\perp) \cup$ { pending: $a_0 \leftarrow$ add1 $(\langle a,0\rangle)$; $a_0 \leftarrow$ stop $(\langle a,0\rangle)$; a_0 [ ] }

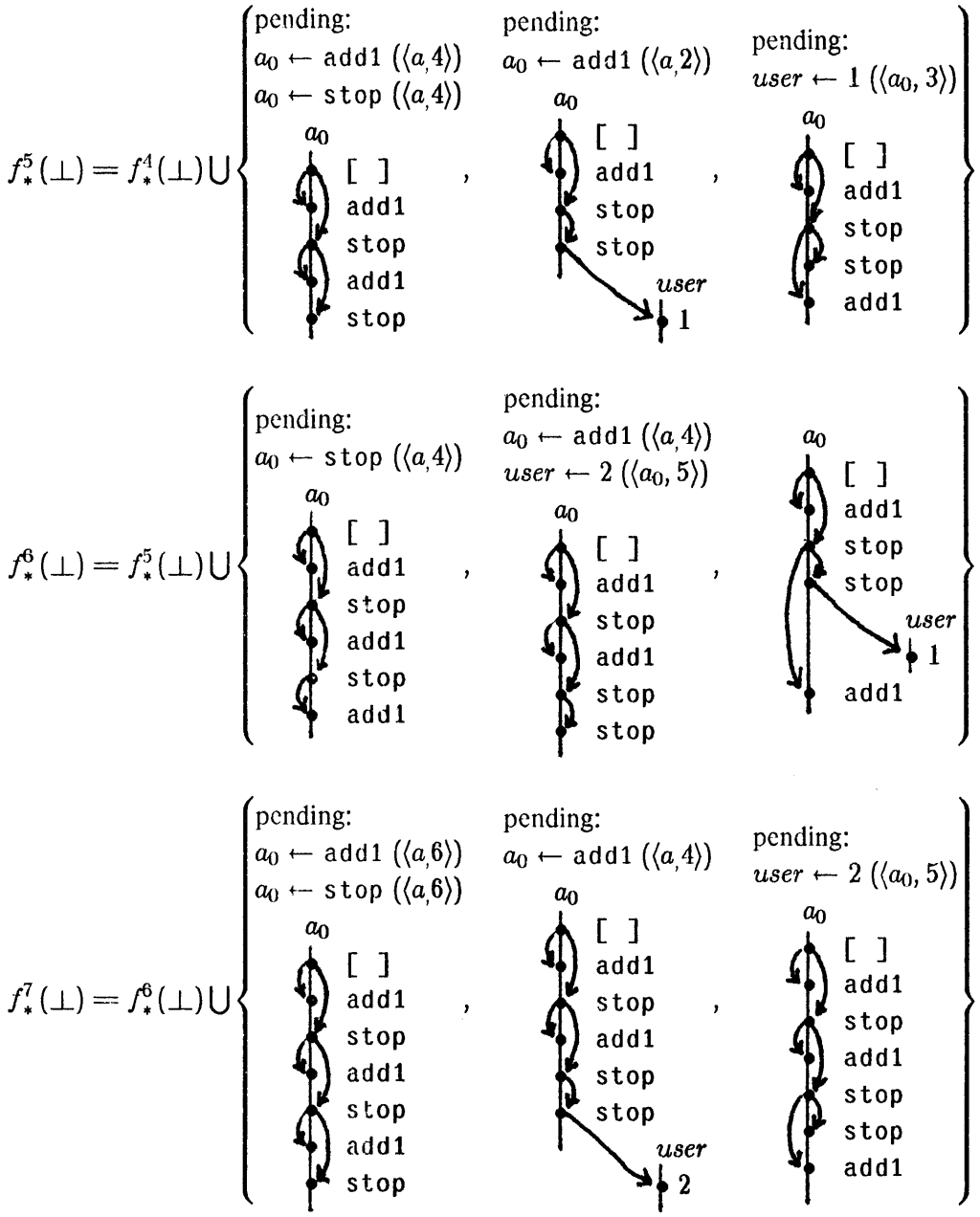
$f_*^2(\perp) = f_*^1(\perp) \cup$ { pending: $a_0 \leftarrow$ stop $(\langle a,0\rangle)$; a_0 [ ] add1 , pending: $a_0 \leftarrow$ add1 $(\langle a,0\rangle)$ $user \leftarrow 0$ $(\langle a_0, 1\rangle)$; a_0 [ ] stop }

$f_*^3(\perp) = f_*^2(\perp) \cup$ { pending: $a_0 \leftarrow$ add1 $(\langle a,2\rangle)$ $a_0 \leftarrow$ stop $(\langle a,2\rangle)$; a_0 [ ] add1 stop , pending: $a_0 \leftarrow$ add1 $(\langle a,0\rangle)$; a_0 [ ] stop *user* 0 , pending: $user \leftarrow 0$ $(\langle a_0, 1\rangle)$; a_0 [ ] stop add1 }

$f_*^4(\perp) = f_*^3(\perp) \cup$ { pending: $a_0 \leftarrow$ stop $(\langle a,2\rangle)$; a_0 [ ] add1 stop add1 , pending: $a_0 \leftarrow$ add1 $(\langle a,2\rangle)$ $user \leftarrow 1$ $(\langle a_0, 3\rangle)$; a_0 [ ] add1 stop stop , a_0 [ ] stop *user* 0 add1 }

(Continued on next page.)

Figure 14. $f_*^i(\perp)$ for choice-loop.



and so on.

Figure 14. (Continued from previous page.) $f_*^i(\perp)$ for `choice-loop`.

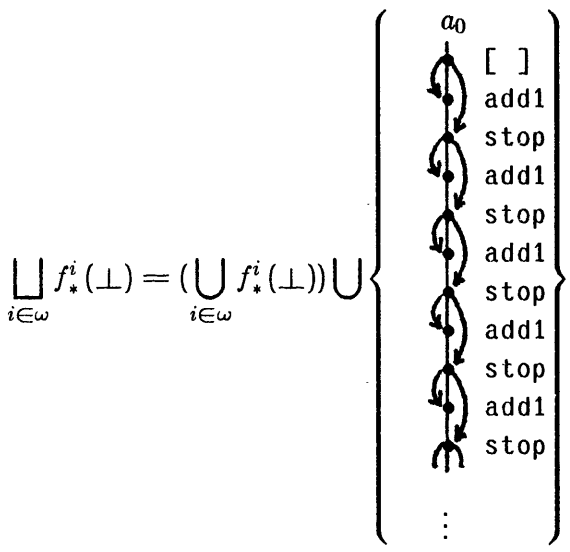


Figure 15. The least fixed point $\bigsqcup_{i\in\omega} f_*^i(\perp)$ for `choice-loop`.

is shown in Figure 16. There is one element in the least fixed point representing a nonterminating computation. Therefore the nondeterminism of the `choice-loop` program is bounded.

One might argue in defense of choice nondeterminism that if choice probabilities are positive, and choices are independent, then programs such as `choice-loop` should terminate with probability 1. Equivalently there should be merge programs that almost always performed a fair merge, in the sense that the probability of an unfair merge would be zero. Such a program would be good enough for engineering purposes. This argument fails because the nondeterminism that appears in a programming language semantics is loose nondeterminism. Implementations are not required to preserve all the nondeterminism that is present in the semantics. In particular, implementations are free to choose the same alternative in every case, so that in some implementations `choice-loop` is certain not to halt.

IV.8. Relation to Standard Power Domains.

Usually in a power domain semantics the least fixed point consists only of completed elements, so applying an operation such as "completed" to the least fixed point is unnecessary. It is tempting

completed$(\bigsqcup_{i\in\omega} f_*^i(\bot)) =$

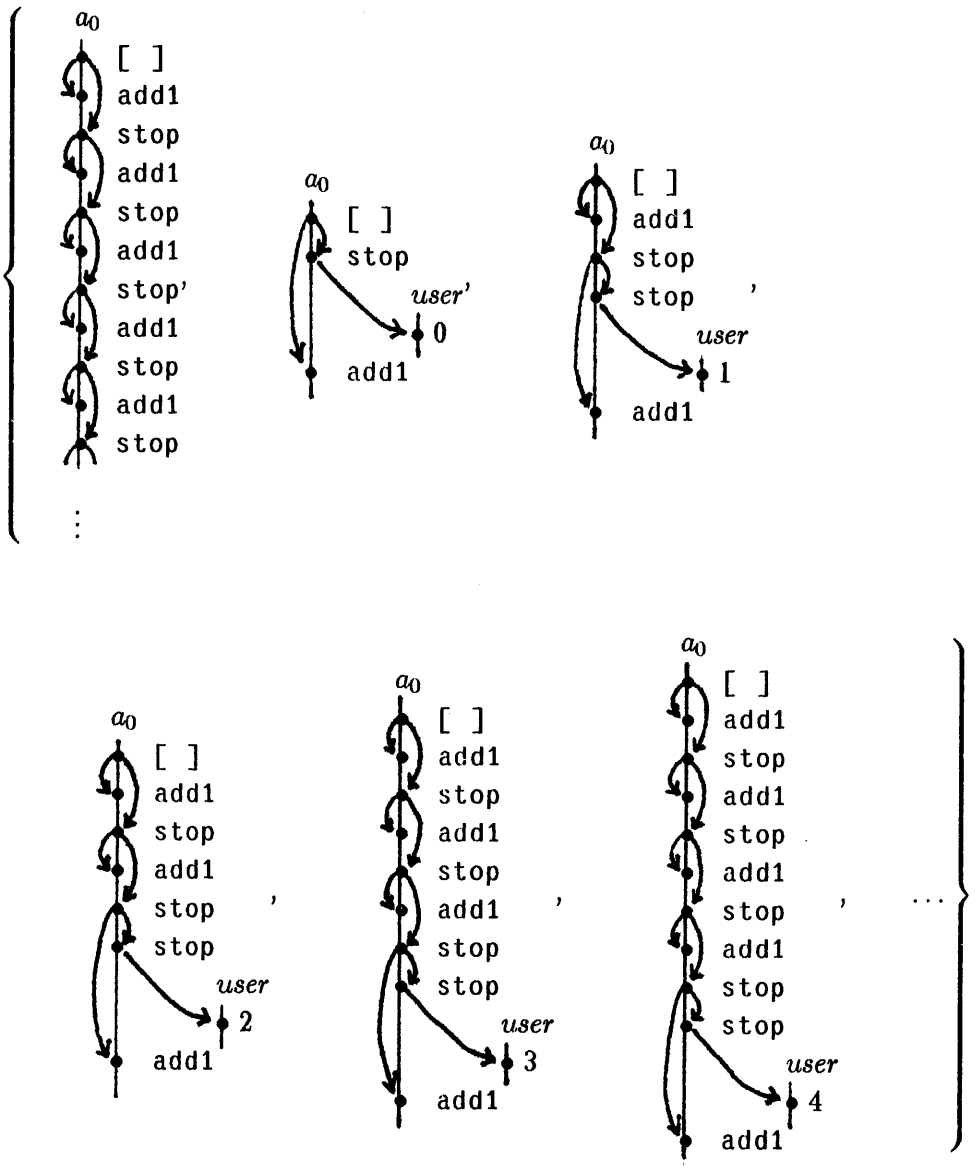


Figure 16. completed$(\bigsqcup_{i\in\omega} f_*^i(\bot))$ for `choice-loop`.

to see this as a defect of the actor power domain, but the quibble can be met on its own terms by the same sort of mathematical sleight of hand that causes least fixed points in other power domains to contain only completed elements. For $\langle D, \leq \rangle$ a domain, define the *frontier* of $A \in P[D]$ to be

$$\mathit{frontier}(A) = \{ x \in A \mid \forall y \in A \; x \leq y \Rightarrow x = y \}.$$

Define the *frontier closure*[15] of $A \subseteq D$ to be

$$A^f = \begin{cases} \mathit{frontier}(A) & \text{if } A^c = (\mathit{frontier}(A))^c; \\ A^c & \text{otherwise.} \end{cases}$$

Then define another power domain $\langle P'[D], \sqsubseteq' \rangle$ by

$$P'[D] = \{ A^f \mid \bot \in A \subseteq D \}$$

and for all $B, C \in P'[D]$

$$B \sqsubseteq' C \quad \Leftrightarrow \quad B^c \subseteq C^c.$$

$\langle P'[D], \sqsubseteq' \rangle$ is clearly isomorphic to $\langle P[D], \sqsubseteq \rangle$ via $A^f \leftrightarrow A^c$.

Consistently replacing references to $P[\mathbf{D}]$, $\sqsubseteq$, $\bigsqcup$, and ^{c} in §4 by references to $P'[\mathbf{D}]$, $\sqsubseteq'$, $\bigsqcup'$, and ^{f} defines f_* as a continuous function from $P'[\mathbf{D}]$ to $P'[\mathbf{D}]$. Its least fixed point is precisely the set of elements of $\mathbf{D}$ that are complete with respect to the initial behaviors.

What then is the relationship between standard power domains and the power domains used here? When D is ω-complete, $P[D]$ is just the standard power domain of D. Chapter III simply extends the standard power domain construction to apply to incomplete domains. This chapter illustrates the value of that extension.

For every domain D the power domain $P[D]$ is isomorphic to the power domain $P[\overline{D}]$ of its ω-completion $\overline{D}$. Nonetheless for some domains $P[D]$ can represent unbounded nondeterminism while $P[\overline{D}]$ cannot. The key to this seeming paradox is that the concrete interpretation placed upon elements of the power domain is important. The purpose of taking fixed points in the power domain is not to select a member of an abstract algebraic structure but to define a subset of D.

[15] The frontier closure is a closure operation on the power set of D with respect to the preorders $\sqsubseteq'$ and $\sqsubseteq_1$ but not with respect to $\subseteq$.

Chapter V

Locality Laws

The locality laws postulated by Hewitt and Baker enforce the idea that all information flow between actors is by means of message passing. As a practical matter, the locality laws rule out side effects to shared environments. Furthermore the information contained in a newly created actor's environment must be a subset of the information in the environment of the actor that created it. The locality laws state these restrictions in a fairly abstract way. They are independent of the ordering laws inasmuch as they further restrict the set of actor event diagrams.

This chapter extends the semantics of Chapter IV to deal with actor creation. It gives an example of a programming language semantics that violates the locality laws. The chapter closes by suggesting that the locality laws ought to be verifiable for the formal semantics of true actor-based languages.

V.1. Actor Acquaintances

In the terminology of programming languages, a procedural object created by associating values with the free variables of a syntactic representation of the procedure is called a closure. Closures are implemented as a pair of pointers, one pointing to the *code* to be executed when the closure is invoked

and the other pointing to the *environment* in which the procedure was closed. The environment supplies values for the free variables.

Actors are analogous to closures. A difference between actors and the objects usually called closures is that closures can share environments, causing *side effects* when one closure changes the environment of another closure. An actor amounts to a closure whose environment is protected from such side effects.

Just as a closure consists of code and an environment, an actor consists of a *script* and a *vector of acquaintances*. The script is simply the code for the actor. The vector of acquaintances provides an environment in which the script is evaluated when the actor accepts a message. An actor's vector of acquaintances can be altered only by that actor.

The vector of acquaintances may contain pointers to other actors. While the pointers themselves cannot be side effected, the behaviors of the actors pointed to can change when those actors process messages sent to them. The vector of acquaintances therefore provides only one level of protection against side effects.

An actor's vector of acquaintances may contain values other than pointers to other actors, or it may consist solely of pointers. In either case the actors that it points to are called *acquaintances* of the actor. An actor may alter its vector of acquaintances while processing a message, so its set of acquaintances may change over time.

V.2. Actor Creation

In statically scoped languages such as Algol and Scheme[1] closures are created by evaluating a procedure abstraction. The environment in effect when the abstraction is evaluated becomes the environment associated with the closure. In actor-based languages actors are created by evaluating a behavior abstraction. The identifier bindings in effect when the abstraction is evaluated are gathered together into a vector of acquaintances. If need be, bindings are copied to protect them against side effects.

[1]Guy Lewis Steele Jr and Gerald Jay Sussman, "The revised report on Scheme: a dialect of Lisp", MIT Artificial Intelligence Memo 452, January 1978.

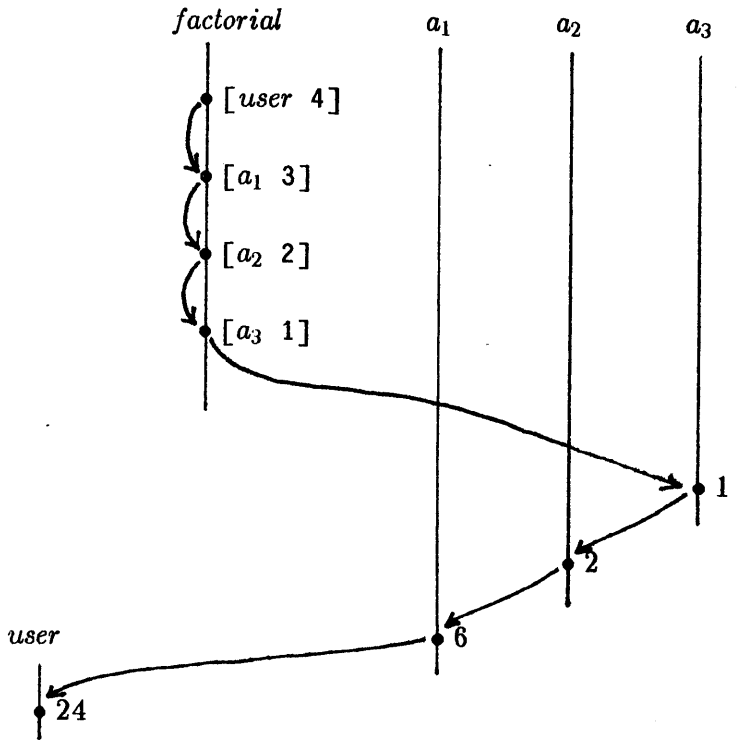


Figure 1. Recursive computation of 4!.

Consider the following subprogram, which computes the factorial function.

```
(factorial = accept [ continuation n ]
             if or [ (lessp [ n 1 ]) (equal [ n 1 ]) ]
                then send 1 to continuation
                else (create ((multiply-by-n
                                 = accept [ x ]
                                   send times [ n x ] to continuation))
                       send [ multiply-by-n (minus [ n 1 ]) ]
                         to factorial))
```

The toy language in which this subprogram is written was designed to make actor creation explicit. If this subprogram is sent the message

```
[ user 4 ]
```

it will create three new actors, a_1, a_2, and a_3, before the result, 24, arrives at the *user* actor. Actor a_1 is created as a result of the first event in the arrival ordering of `factorial`, a_2 is created as a result of the second event, and a_3 is created as a result of the third event. The event diagram for the computation is shown in Figure 1.

The three created actors share the script

```
accept [ x ]
send times [ n x ] to continuation
```

This script has two free identifiers, `n` and `continuation`. When `factorial` accepts the message `[ user 4 ]`, it binds `continuation` to *user* and `n` to 4, and those are the bindings in effect when the `create` command is first encountered, so the vector of acquaintances for a_1 is

identifier	value
n	4
continuation	⟶ *user*

`factorial` then sends itself the message `[` a_1 `3 ]`, so the vector of acquaintances for a_2 is

identifier	value
n	3
continuation	⟶ a_1

The last actor to be created is created while `n` is bound to 2 and `continuation` is bound to a_2, so the vector of acquaintances for a_3 is

identifier	value
n	2
continuation	⟶ a_2

These vectors of acquaintances may be kept on a stack in a sequential implementation. They are part of the actor conceptually, however.

The distinction between iterative and recursive programs can be easily expressed in the actor model: iterative programs do not create any new actors.[2] The following tail-recursive program, for example, is iterative.

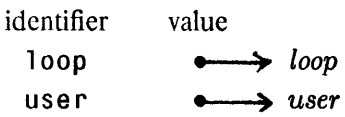
```
(factorial = accept [ continuation n ]
             send [ continuation n 1 ] to loop)

(loop = accept [ continuation n product ]
        if or [ (lessp [ n 1 ]) (equal [ n 1 ]) ]
           then send product to continuation
           else send [ continuation
                       (minus [ n 1 ])
                       (times [ n product ]) ]
                to loop)
```

The actors created by the recursive version of `factorial` never change their vectors of acquaintances. For an example of an actor that changes its vector of acquaintances, consider the `loop` program of §IV.5:

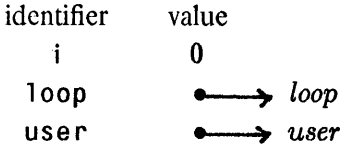
```
(loop = accept [ ]
        send "add1" to loop ;
        become i initially 0
                 inside
               accept [ msg ]
               if equal [ msg "add1" ]
                  then change i to plus [ i 1 ] ;
                       send "add1" to loop
                  else
               if equal [ msg "halt" ]
                  then send i to user ; become accept [ ] dummy
                  else dummy).
```

Its vector of acquaintances starts out with two entries, one of which points to `loop` itself.

[2]Carl Hewitt, "Viewing control structure as patterns of passing messages", *Artificial Intelligence* 8, 1977, pages 323-363.



Upon accepting its first message `loop` adopts a new behavior, differing in both the script and in the vector of acquaintances:



It then proceeds to increase the value of i each time it accepts an `add1` message.

V.3. Locality Laws Add Power

Hewitt and Baker[3] have proposed *locality laws* stating reasonable restrictions on the set of acquaintances of an actor and relating acquaintances to actor event diagrams. This section gives a variant of the locality laws and shows that adding the locality laws to the ordering laws considered in Chapter II gives a more powerful theory.

To the structure

$$\langle \mathrm{E}, \mathrm{A}, T, -act\rightarrow, \mathrm{Arr} \rangle$$

considered in Chapter II and consisting of the set of events, the set of actors, the target function, the activation ordering, and the set of arrival orderings, add three new objects *acq*, A_0, and *creation* to obtain a structure

$$\langle \mathrm{E}, \mathrm{A}, T, -act\rightarrow, \mathrm{Arr}, acq, \mathrm{A}_0, creation \rangle.$$

acq is a function: $\mathrm{E} \rightarrow$ subsets(A) giving for each event e the set of acquaintances of $T(e)$ at the time of the event e. Intuitively $acq(e)$ is the set of actors that the target of e already knew about

[3]"Laws for communicating parallel processes", IFIP-77, Toronto, August 1977, pages 987–992, and "Actors and continuous functionals", IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, 16.1–16.21.

when it accepted the message of e. A_0 is the set of *primeval* actors, the set of actors that exist when computation begins. Thus A_0 is a finite subset of A. *creation* is a function: $(A - A_0) \rightarrow E$ giving for each actor created in the course of computation the event that caused its creation.

Hewitt and Baker stated the locality laws in terms of a fourth new object, the *participants* in an event. The participants in an event are those actors that the target of the event knows about while processing the message of the event. The participants are thus the acquaintances of the target together with the actors mentioned by the message.

For an external event, the message can mention an arbitrary finite set of actors, so there is no restriction on the participants of an external event except that they form a finite set.[4] For events that are not external, though, the participants must come from among the acquaintances of the target of the event, the actors created by the event, and the participants in the activator of the event.

Rather than introduce the *participants* function into the structure, this section treats it like global time and simply asserts the existence of a function with the required properties. The locality laws then become

Law of Finite Acquaintances. *acq* (e) *is finite for every* $e \in E$.

Existence of Participants Function. *There exists a function participants*: $E \rightarrow$ *subsets*(A) *satisfying the following laws.*

Finite Interaction Law. *participants* (e) *is finite for every* $e \in E$.

Let $created\,(e) = \{\, a \in A - A_0 \mid creation\,(a) = e \,\}$.

Original Acquaintances Law. *If a is a created actor, that is,* $a \notin A_0$, *and e is the first event in the arrival ordering of a, then*

$$acq\,(e) \subseteq participants\,(creation\,(a)) \bigcup created\,(creation\,(a)).$$

[4]Perhaps there should be a restriction that the message of an external event can mention only primeval actors.

Arrival Precursor Acquaintances Law. *If* $a = T(e)$ *and* e *has an immediate predecessor* e' *in the arrival ordering of* a*, that is,* $e' \text{—}arr_a\text{→} e$ *and* $\neg\exists e''\ e' \text{—}arr_a\text{→} e'' \text{—}arr_a\text{→} e$*, then*

$$acq(e) \subseteq participants(e') \bigcup created(e').$$

If $e \in \mathrm{E}$ is not external, then let $activator(e)$ be the activator of e, that is, the unique immediate predecessor of e in the activation ordering $\text{—}act\text{→}$.

Activator Acquaintances Law. *If* $e \in \mathrm{E}$ *is not external then*

$$T(e) \in participants(activator(e)) \bigcup created(activator(e))$$

$$participants(e) \subseteq acq(e) \bigcup participants(activator(e)) \bigcup created(activator(e)).$$

The second half of the last law differs somewhat from Hewitt and Baker's formulation.

The first half of the Activator Acquaintances Law relates the locality laws to the actor event diagrams. Adding the locality laws to the ordering laws produces a more powerful theory, as shown by the following actor event diagram which satisfies all the ordering laws of Chapter II but is ruled out by the locality laws.

The actor event diagram is shown in Figure 2. The idea is that two actors a and a' never communicate with each other, so they can have only a finite amount of information in common, but each sends messages to the same infinite set of actors. That cannot be, because there is no way the same infinite set of pointers to actors can pass through both of a and a'.

Formally the actor event diagram of Figure 2 is described by the structure

$$\langle \mathrm{E}, \Lambda, T, \text{—}act\text{→}, \mathrm{Arr} \rangle$$

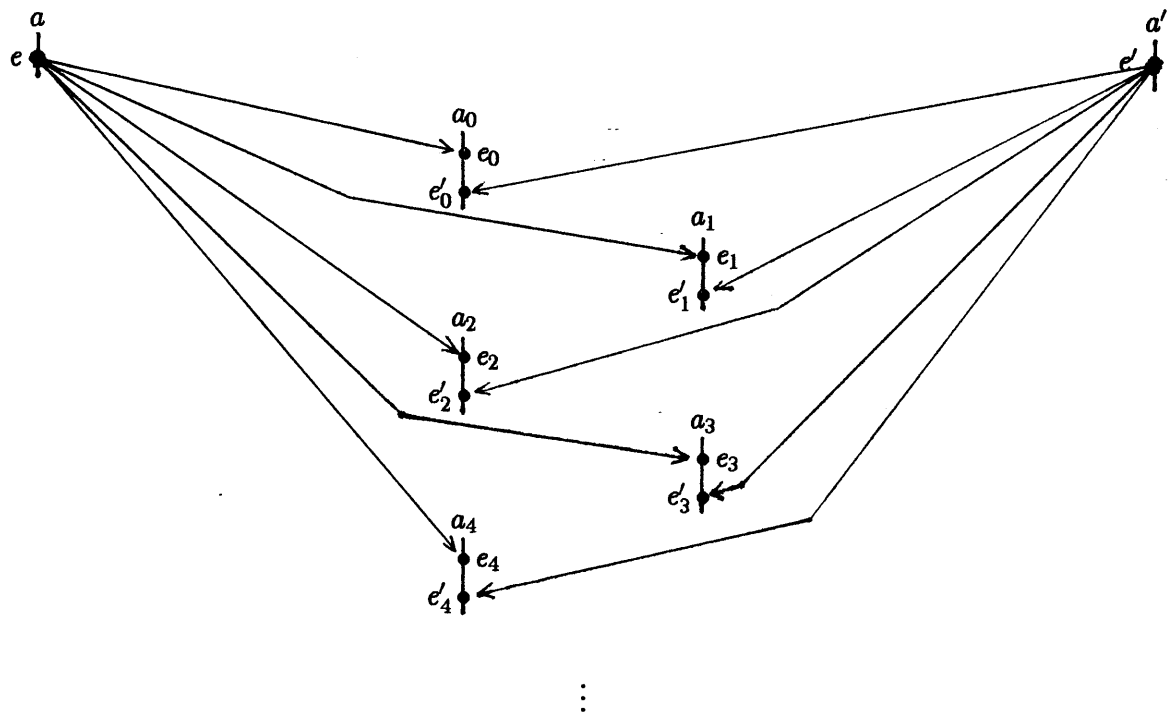


Figure 2. An actor event diagram that violates the locality laws.

where

$$\mathrm{E} = \{e, e'\} \bigcup \{e_i \mid i \in \omega\} \bigcup \{e'_i \mid i \in \omega\}$$
$$\mathrm{A} = \{a, a'\} \bigcup \{a_i \mid i \in \omega\}$$
$$T(e) = a$$
$$T(e') = a'$$
$$e_i \text{ —}arr_{a_i}\text{→ } e'_i \quad \text{for all } i \in \omega$$
$$e, e' \text{ are external events}$$
$$e \text{ —}act\text{→ } e_i \quad \text{for all } i \in \omega$$
$$e' \text{ —}act\text{→ } e'_i \quad \text{for all } i \in \omega$$

This structure satisfies the ordering laws of Chapter II, yet there is no way to extend it to a structure

$$\langle \mathrm{E}, \mathrm{A}, T, \text{ —}act\text{→}, \mathrm{Arr}, acq, \mathrm{A}_0, creation \rangle$$

satisfying the locality laws. Proof: suppose there were such an extension, with a given *participants* function satisfying the locality laws. Then *participants*(e) and *participants*(e') are both finite,

so their union is also finite. Let $n \in \omega$ be such that a_n is not in their union. e and e' are the only external events, so the Activator Acquaintances Law applies at both e_n and e'_n. Furthermore $e = activator(e_n)$ and $e' = activator(e'_n)$. Hence $creation(a_n) = e$ and $creation(a_n) = e'$, a contradiction.

This actor event diagram can be modified so that there is only one external event and no event activates infinitely many events, and a similar proof will still go through.

§II.7 showed that the ordering laws were independent of the locality laws. This section has returned the favor by showing that the locality laws are independent of the ordering laws.

V.4. Semantics with Actor Creation

Chapter IV gave a power domain semantics for actor-based languages without actor creation. This section extends the semantics of Chapter IV to permit actors to be created during the course of computation.

The concept of programming language semantics that has the most to do with the technical adjustments in this section is the concept of a *store*. Usually a store is a mapping from locations to stored values. Here it will be a mapping from actor names, or network addresses, to behaviors. Usually updated versions of the store are passed from semantic function to semantic function. Here and in Chapter IV the original store is passed together with enough history to reconstruct the updated store. Usually the question of exactly which unused location is pressed into service when a new object is created is left unanswered by programming language semanticists. On this question, and often on this question only, semanticists usually resort to axioms rather than give a concrete denotation.[5] Here a concrete answer will be given to the question of which unused actor name should be allocated to a new actor. However, the set of actor names will not bear any resemblance to the space of network addresses for real machines. The correspondence between actor names and network addresses is to be determined by the storage management module in real implementations.

[5]See for example the discussion of *new* in §1.4.2 of Milne and Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London, 1976.

The semantics given in §IV.4 begins by assuming a function

$$\mathcal{P}(Q)\colon \mathsf{A} \rightarrow \mathsf{F}$$

giving the initial behavior of each actor. The obvious way to extend the semantics to deal with actor creation is to let $\mathcal{P}(Q)$ specify only the behaviors of primeval actors and to let the semantics keep track of the behavior of a created actor beginning with the time of its creation. This approach is sound, would work, and is the approach usually taken, but it would require significant revisions to the power domain semantics of Chapter IV. The revisions would be necessary because of a shortcut that was taken to simplify the semantics. The semantics in Chapter IV does not associate a mapping from actors to current behaviors with each event diagram. Rather it computes current behaviors from the initial behaviors and the initial history provided by an event diagram.

This section instead makes $\mathcal{P}(Q)\colon \mathsf{A} \rightarrow \mathsf{F}$ give the initial behavior of every actor, primeval and created alike, that could possibly exist during a computation. That is accomplished through the inelegant technical trick of coding within the name of each created actor a pointer to its creation event. Indeed a created actor's name will include the entire local history of the actor that created it, up to and including its creation event. To be specific, the set of actor names is defined by the reflexive domain equation

$$\mathsf{A} = \{user\} + (\{program\} \times \mathsf{N}) + ((\mathsf{A} \times \mathsf{M}^+) \times \mathsf{N})$$

where *user* and *program* are distinct atomic symbols, N is the flat domain of natural numbers, and M^+ is the domain of nonempty sequences of messages. The interpretation of the actor names is as follows.

user is one of the primeval actors. It is meant to denote a terminal, file, or operating system through which programs can communicate results to their user.

$\langle program, 0\rangle$ is the first actor declared in a program, so it too is a primeval actor. In general $\langle program, \nu\rangle$ is the $(\nu + 1)$th of the primeval actors declared in a program. All actors of the form $\langle program, \nu\rangle$ are primeval if they exist.

$\langle\langle \alpha, \mu^*\rangle, \nu\rangle$ is the name of the $(\nu + 1)$th actor created as a result of the nth event in the arrival ordering of α, where n is the length of the sequence μ^*. The ith element of μ^* is the message of the

ith event in the arrival ordering of a. Thus μ^* codes the local history of a that led to the creation of $\langle\langle a, \mu^*\rangle, \nu\rangle$. Note that if a is itself a created actor then the name 'a' points to *its* creation event, and so on. In this way every actor name traces history all the way back to a primeval actor, making possible an inductive definition of $\mathcal{P}(Q)$ with the primeval actors as the basis for the induction.

Recall that in §IV.2 the behavior domain was defined via the equation

$$\mathsf{F = M \rightarrow (F \times (A \times M)^*)}$$

so a behavior was a function from messages to pairs consisting of a new behavior and a finite sequence of messages sent to target actors. Allowing actors to create a finite number of new actors upon accepting a message causes the behavior domain to become

$$\mathsf{F = M \rightarrow G \rightarrow (F \times (A \times M)^* \times F^*)}$$

where an element of F^* is a finite sequence of behaviors—the initial behaviors of the created actors. An element of G is an actor name generator producing the names to be given to the actors created in an event. The domain G is defined by

$$\mathsf{G = A \times G.}$$

The only changes that need to be made to §IV.4 to accomodate actor creation are caused by the addition of actor name generators to the behavior domain equation. The semantics must supply behaviors with both a message and the correct actor name generator.

The definition in §IV.4 of the successive behaviors of an actor a must be changed to

$$\begin{aligned} b(a, 0) &= \mathcal{P}(Q)\, a \\ b(a, n+1) &= next\,(b\,(a, n)\,(M\,(\langle a, n\rangle))\,\gamma_{n+1}) \end{aligned}$$

where γ_{n+1} is the actor name generator producing the new actor names

$$\langle\langle a, \mu^*\rangle, 0\rangle,\ \langle\langle a, \mu^*\rangle, 1\rangle,\ \langle\langle a, \mu^*\rangle, 2\rangle,\ \ldots.$$

Here μ^* is a list of the first $n+1$ messages to arrive at a. Thus

$$\gamma_{n+1} = gamma\,(\langle a, history\,(a, n+1)\rangle)$$

where *gamma* is as defined in the appendix and *history* is defined by

$$history\,(a, 0) = \langle\,\rangle$$
$$history\,(a, n+1) = (history\,(a, n)) \S \langle M\,(\langle a, n\rangle)\rangle$$

where § indicates concatenation of sequences.

In the definition of $g\,(\perp)$ in §IV.4 the pending events P must be changed to

$$P = pend\,(\mathcal{P}\,(Q)\,a_0\,m_0\,\gamma) \times \{\,\langle a_0, 0\rangle\,\},$$

where

$$\gamma = gamma\,(\langle a_0, \langle m_0\rangle\rangle)$$

while in the definition of $x\,(a, m, e)$ the pending events P' must be changed to

$$P' = (P - \{\,\langle\langle a, m\rangle, e\rangle\,\}) \biguplus (pend\,(behavior\,(a)\,m\,\gamma) \times \{\,\langle a, n\rangle\,\})$$

where

$$\gamma = gamma\,(\langle a, (history\,(a, n)) \S \langle m\rangle\rangle).$$

In the definitions of augmented actor event diagrams consistent with respect to the initial behaviors and complete with respect to the initial behaviors the left hand side of the main equation must be changed from

$$pend\,(b\,(a, n)\,(M\,(e)))$$

to

$$pend\,(b\,(a, n)\,(M\,(e))\,\gamma)$$

where

$$\gamma = gamma\,(\langle a, history\,(a, n+1)\rangle).$$

The theorems of §IV.4 are unaffected by these technical changes. The changes make possible a definition of $\mathcal{P}\,(Q)$: $\mathsf{A} \rightarrow \mathsf{F}$ giving an initial behavior for all actors that could possibly be created during computation. The appendix contains the details.

By way of apology I would like to quote Milne and Strachey:[6]

> In situations where any one of a large number of models is equally satisfactory it might well seem better to give a set of axioms which all the models need to satisfy and to refrain from making the extra and arbitrary choices any particular model involves. We shall not adopt this course, because the use of a particular model allows us to give our results a more concrete form and, we think, improves the intelligibility of an already complex subject.

Readers who feel that the treatment of actor names in this section is a counterexample to that argument have my sympathy.

V.5. A Toy Language

A dissertation on defining the semantics of actor-based programming languages ought to define the semantics of an actor-based programming language. The appendix presents the semantics of a toy language illustrating actors, culminating in a function

$$\mathcal{P}: \mathtt{Act} \rightarrow (\mathtt{A} \rightarrow \mathtt{F})$$

giving for each program in `Act` an assignment of initial behaviors to actors. At that point the power domain semantics of Chapter IV takes over.

The toy language presented in the appendix, dubbed Atolia for ease of reference, was designed expressly to illustrate this dissertation. It is a horrid programming language, as the sample programs in the appendix demonstrate. The one thing Atolia does well is reflect the semantics of message passing and actor creation.

An interpreter for Atolia programs has been written in Lisp for the DEC PDP-10 and the Lisp Machines at the MIT Artificial Intelligence Laboratory. The interpreter normally runs programs pseudo-concurrently and is nondeterministic. Efficiency was not a concern when the interpreter was built. Comparisons made on the PDP-10 show that Atolia programs run three to seven times slower

[6] *ibid.*

than comparable Scheme[7] programs. Implementation and testing of the interpreter took ten person-days. The Atolia programs contained in this dissertation were tested using the interpreter.

V.6. The Locality Laws may not Hold

Do the locality laws hold for Atolia? The semantic definition of Atolia given in the appendix does not answer that question because the definition is incomplete. The semantic function

$$\mathcal{O}: \mathsf{Opr} \rightarrow \mathsf{V} \rightarrow \mathsf{V}$$

giving the meaning of primitive operators is not defined. If Atolia has sufficiently strange primitive operators, then the locality laws do not hold. Let $a_0, a_1, a_2, \ldots$ be distinct actors, and consider the function *strange*: V $\rightarrow$ V defined by

$$strange(\epsilon) = \begin{cases} a_0 \text{ in } \mathsf{V} & \text{if } \epsilon = true \text{ in } \mathsf{V}; \\ a_{i+1} \text{ in } \mathsf{V} & \text{if } \epsilon = a_i \text{ in } \mathsf{V}; \\ \epsilon & \text{otherwise} \end{cases}$$

where (a in V) is the injection of a into the domain V. If Atolia contains a primitive operator `strange` such that

$$\mathcal{O}[\![\texttt{strange}]\!] = strange$$

then the locality laws do not hold. The reason is the primitive operator `strange` makes it possible for an actor to send messages to an infinite set of actors without ever creating an actor or accepting a message from any actor other than itself. Consider the program

[7]Guy Lewis Steele Jr and Gerald Jay Sussman, "The revised report on Scheme: a dialect of Lisp", MIT Artificial Intelligence Memo 452, January 1978.

```
(startup = accept [ ]
           send strange(true) to A ;
           send strange(true) to B)

(A = accept [ actor ]
     send "greetings" to actor ;
     send strange(actor) to A)

(B = accept [ actor ]
     send "greetings" to actor ;
     send strange(actor) to B)
```

This program does not heed the locality laws. The actor event diagrams that correspond to its computations resemble the actor event diagram proved in §3 to violate the locality laws.

The effect of the locality laws is to rule out such strange primitive operators. To put it differently, the locality laws call on a semantics to account for such operators in terms of message passing and actor creation so that they no longer appear as primitives. The point is that the locality laws do not automatically hold for a programming language semantics. A semantics for which the locality laws fail may be perfectly acceptable for some purposes, but it is not a true actor semantics.

V.7. The Locality Laws may be Provable

The previous section showed that if the primitive operators of Atolia are ill behaved, then the locality laws do not hold. If on the other hand the primitive operators are well behaved, then the locality laws do hold for Atolia.

This claim has the status of a conjecture rather than a proved theorem. Its proof would involve a structural induction encompassing every semantic equation in the appendix, and that structural induction has not been carried out. Nonetheless a compelling plausibility arguement can be based on a simple inspection of those equations.

The value domain of Atolia is

$$V = T + N + R + H^* + A + V^*$$

where A is the domain of actors, V* is the domain of sequences of values, and the other domains are basic domains not involving actors.[8] Define the set of actors embedded in a value $\epsilon \in \mathsf{V}$ as $s(\epsilon)$ where

$$s(\epsilon) = \begin{cases} \emptyset & \text{if } \epsilon \in \mathsf{T}; \\ \emptyset & \text{if } \epsilon \in \mathsf{N}; \\ \emptyset & \text{if } \epsilon \in \mathsf{R}; \\ \emptyset & \text{if } \epsilon \in \mathsf{H}^*; \\ \{a\} & \text{if } \epsilon \in \mathsf{A} \text{ and } \epsilon \mid \mathsf{A} = a; \\ \emptyset & \text{if } \epsilon \in \mathsf{V}^* \text{ and } \epsilon \mid \mathsf{V}^* = \langle\,\rangle; \\ \bigcup_{i=0}^{n} s(\epsilon_i) & \text{if } \epsilon \in \mathsf{V}^* \text{ and } \epsilon \mid \mathsf{V}^* = \langle \epsilon_0, \ldots, \epsilon_n \rangle. \end{cases}$$

(Here $\epsilon \mid D$ is the projection of ϵ to the domain D.) A primitive operator $\mathsf{O} \in \mathsf{Opr}$ is *well behaved* iff for all $\epsilon \in \mathsf{V}$

$$s(\mathcal{O}[\![\mathsf{O}]\!]\epsilon) \subseteq s(\epsilon)$$

so that applying the operator to a value produces a result value embedding only actors that were already present in the argument value. If every Atolia operator is well behaved in this sense, then the locality laws hold.

Idea of proof: it should be clear how to define the primeval actors A_0 and the creation function *creation* for a computation performed by an Atolia program. There are several ways to define the acquaintances function *acq*. The simplest way is to define

$$acq(e) = \bigcup_{\mathsf{I} \in \mathsf{Ide}} s(\rho[\![\mathsf{I}]\!])$$

where ρ is the environment giving the values of identifiers appearing in the script of $T(e)$ at the time of the event e. An alternative is to take the union only over those identifiers appearing free in the script of $T(e)$. Both definitions serve the purpose. From either one a *participants* function can be defined by

$$participants(e) = \begin{cases} acq(e) & \text{if } e \text{ is external;} \\ acq(e) \bigcup participants(e') \bigcup created(e') & \text{if } e' = activator(e). \end{cases}$$

[8] In Act1 the value domain is V = A because everything is an actor. Act1 does not have primitive operators, but has primitive actors, which Atolia does not have. With a few changes mandated by those differences the remarks of this section would apply equally to proving the locality laws for Act1.

participants (e) is thus the set of all actor names that could possibly be accessible to the target of e while it is processing the message of e. Defining *participants* $(e) = acq\,(e)$ for external events works only because the single external event of an Atolia computation mentions no actors. If the message of an external event could mention actors, then those actors would have to be included among the participants.

The second half of the Activator Acquaintances Law is immediate from the definition of *participants*.

Since only finitely many identifiers are bound in the initial environment, the identifier binding mechanisms of Atolia bind only finitely many identifiers at a time, and Atolia scripts always terminate, only finitely many identifiers can become bound as the result of an event. Furthermore *created* (e) is always finite. An induction on the number of predecessors of an event in the combined ordering thus proves both the Law of Finite Acquaintances and the Finite Interaction Law.

Yet to be established are the Original Acquaintances Law, the Arrival Precursors Acquaintances Law, and the first half of the Activator Acquaintances Law. These are the nontrivial locality laws. They all depend upon the idea that the only way an actor name can become known to an actor a is by being present in the environment prevailing when a is created, by being part of a message sent to a, or by being the name of an actor created by a. Proving the locality laws for Atolia amounts to verifying this idea from the semantic equations given in the appendix.

Inspection reveals that the only possible problem is the primitive operators. So long as they are well behaved, though, an actor cannot use them to come up with any new actor names that the actor doesn't already know about. If the primitive operators are well behaved, therefore, the locality laws hold.

Chapter VI

Conclusion

This thesis has set forth the foundations of a theory of semantics for nondeterministic programming languages based on the actor model of concurrent computation. To that end, the thesis has given a precise account of the actor model. It has justified the ordering laws using a notion of global time realizability. It has demonstrated a constraining effect of the locality laws. It has analyzed notions of concurrency and nondeterminism. It has extended a standard power domain construction to apply to incomplete domains, and has used that extension to define a power domain semantics for actor-based languages.

The actor semantics presented in this thesis is not very abstract because the event diagrams contain far too much operational information for most purposes. For example, the Atolia program

```
(f = accept [ ] send [ ] to g)
(g = accept [ ] send 0 to user)
(h = accept [ ] dummy)
```

does not have the same meaning as

```
(f = accept [ ] send [ ] to g)
(h = accept [ ] dummy)
(g = accept [ ] send 0 to user)
```

because the second actor to be declared receives a message in the first program but not in the second. This is analogous to a problem that arises in standard semantics when two programs that are intuitively equivalent turn out to have different meanings because they use storage in slightly different ways.[1] In standard semantics the problem is made much less severe by concentrating on the final output of a program. In actor semantics it is not clear what should be considered the final output, though often the only thing of importance is the arrival ordering of a particular actor such as ***user***. This matter deserves further attention.

The semantics presented in this thesis needs to be extended to other kinds of actors besides primitive serializers. One goal of this extension should be to make it possible to regard a complex system of actors as a single actor.

The technique of building power domains from incomplete domains is not limited to actor semantics. A fair power domain semantics for dual processors communicating via shared memory can also be constructed using this technique. I conjecture that an incomplete history domain could be used to construct a fair power domain semantics for the language of Communicating Sequential Processes.

The power domains with incomplete underlying domains that have so far occurred to me seem unpleasantly operational, but the real limitations of the idea are not yet known.

The category of (possibly incomplete) domains and ω-continuous maps as defined in Chapter III is closed with respect to the usual domain constructors +, x, *, $\rightarrow$, and the power domain construction $P[\,\cdot\,]$ of Chapter III. A theorem stating conditions under which reflexive domain equations have solutions in that category would be very useful.

[1]See §4.1.1 of Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London, 1976.

Appendix I

Atolia: Informal Description

This appendix describes the abstract syntax and informal semantics of a toy language illustrating actors.

Atolia

(a toy language illustrating actors)

Version 0

Syntactic domains

$I \in$ Ide	identifiers
$B \in$ Bas	bases
$O \in$ Opr	operators
$E \in$ Exp	expressions
$\Gamma \in$ Com	commands
$\Phi \in$ Abs	abstractions (scripts)
$\Delta \in$ Dec	local declarations
$\Sigma \in$ Act	actor script declarations

Productions

E ::= B | OE | I | $[E_0 \cdots E_n]$
| if E_0 then E_1 else E_2 | (E)

Γ ::= dummy | change I to E | become Φ | send E_0 to E_1
| create (Σ) Γ | Γ_0; Γ_1 | if E then Γ_0 else Γ_1 | (Γ)

Φ ::= accept $[I_0 \cdots I_n]$ Γ | Δ inside Φ | if E then Φ_0 else Φ_1 | (Φ)

Δ ::= I initially E | $I = E$ | Δ_0; Δ_1 | (Δ)

Σ ::= $(I = \Phi)$ | Σ_0 Σ_1

Expressions

Bases

B

The bases are the constants and literals, such as the booleans `true` and `false`, the numerals representing integers such as `0` and `1`, representations for whatever other number types are needed, and character strings such as `"this is a string"`. They evaluate to the basic values of the machine.

Operator applications

OE

An operator application consists of an operator followed by an expression. To simplify the language, all operators take exactly one argument, but the effect of two or more arguments can be obtained by using a sequence as the argument. The expression is evaluated and fed to the operator, which returns a single result value. As is the case for all Atolia expressions, there are no side effects.

Among the operators are predicates and functions such as `equal`, `actorp`, `plus`, and `times`. The operators of Atolia are fixed by the language; users cannot define additional operators.

Identifiers

I

An identifier denotes a basic value, an actor, or a sequence of denoted values. In other words, an identifier can denote the result of any Atolia expression. Identifiers are bound by local declarations, by the patterns of `accept` statements, and by actor script declarations.

Sequences

$[E_0 \cdots E_n]$

A list of expressions in brackets indicates a sequence of values. Since sequences are themselves expressions, sequences may be nested.

Conditional expressions

`if` E_0 `then` E_1 `else` E_2

The expression in the predicate position must evaluate to a boolean value. If it evaluates to *true*, the expression following the `then` is evaluated and becomes the value of the conditional expression; otherwise the expression following the `else` is evaluated and becomes the value of the expression. As with all expressions in Atolia, the predicate expression has no side effects.

Parenthesized expressions

(E)

Parentheses are ignored by the semantic equations. They appear in the abstract syntax to allow syntactically unambiguous programs to be written.

Commands

Dummy commands

`dummy`

The `dummy` command has no effect.

Assignments

`change` I `to` E

The `change` command causes the identifier to denote a new value. The identifier being `changed` must be mutable; in other words, it must have been declared by a declaration of the form I `initially` E.

New behaviors

`become` Φ

The `become` command specifies a new behavior for the actor, to become effective when the actor unlocks. Only a subsequent `become` command can override the newly specified behavior. The free identifiers of Φ are bound to the values they denote when the `become` command is executed. Identifiers that are mutable at the time of the `become` command remain mutable in Φ unless redeclared or bound.

Transmissions

`send` E_0 `to` E_1

The `send` command evaluates expression E_0 and sends the result as a message to the actor specified by E_1. E_1 must evaluate to an actor, of course.

Actor creations

`create` (Σ) Γ

The `create` command is similar to the `letrec` expression of ISWIM and the `labels` expression of Scheme. It permits the creation of mutually recursive actors. First the identifiers denoting

the newly created actors are bound to their newly allocated network addresses. Then the behaviors of the new actors are fixed by binding the free variables of their scripts in the resulting environment. The new actors are not permitted to change the state variables of their creating actor, however, nor do subsequent changes by their creating actor affect the values of identifiers in the new actors. Then the creating actor executes a command before discarding the environment that contains the addresses of the new actors. The command may send messages to the new actors or may change a state variable to remember some of them as new acquaintances; there is no point to a `create` command of the form `(create (Σ) dummy)`.

Sequencing

Γ_0; Γ_1

Γ_0 is executed, followed by Γ_1. Atolia has no gotos or other sequencers that could alter the sequential order of execution.

Conditional commands

`if` E `then` Γ_0 `else` Γ_1

The expression must evaluate to a boolean. If the result is true, Γ_0 is executed; otherwise Γ_1 is executed. The evaluation of the predicate expression has no side effects.

Parenthesized commands

(Γ)

Parentheses are ignored.

Abstractions (Scripts)

Accept statements

`accept` $[I_0 \cdots I_n]$ Γ

The `accept` statement specifies a behavior. In Lisp terms, it is a lambda expression that evaluates to a closure when it is encountered as part of an actor script declaration or `become` statement. When the actor whose behavior it specifies first receives a message, it locks, binds identifiers I_0 through I_n to components of the message, executes the command Γ, and then unlocks. The command Γ may cause messages to be sent and/or actors to be created. Γ also determines a new behavior for the actor. If executing Γ does not result in executing any `become` or `change` commands, the new behavior is the same as the old. If `become` commands are encountered, the last one determines the new behavior of the actor. `change` commands can alter the behavior of an actor by changing the values of mutable identifiers.

The identifiers bind to message components as follows. Usually the message is a sequence, in which case the elements of the message pair one-for-one with the corresponding identifiers, proceeding from left to right. If the message sequence is longer than the list of identifiers, the extra message components are ignored. If the list of identifiers is longer, the extra identifiers bind to the empty sequence. If the message is not a sequence, every identifier in the identifier list binds to the value of the message. If the identifier list is empty, no identifiers are bound and the message acts only to initiate execution of the command Γ. The exact manner in which the identifiers are bound to the message components is to a great extent arbitrary, of course. The language Act1, on which Atolia is based, uses a considerably more sophisticated matcher.

Abstractions governed by local declarations

Δ `inside` Φ

The purpose of a local declaration is to bind identifiers referred to inside an abstraction.

Conditional abstractions

`if` E `then` Φ_0 `else` Φ_1

The expression must evaluate to a boolean. If the result is *true*, then Φ_0 is the abstraction to be used. Otherwise Φ_1 is used.

Parenthesized abstractions

`(`Φ`)`

Parentheses are ignored.

Local Declarations

Mutable declarations

I `initially` E

The expression is evaluated and bound as the value of the identifier.

Identifiers declared using `initially` are similar to "own" variables bound at declaration time. They are state variables of the actor whose abstraction contains their declaration. Only that actor can alter them by `change` commands. When new actors are created, the new actors' scripts may refer to state variables of the creating actor, but the value denoted by those references is fixed as the value of the state variables at the time of the created actors' declarations. Not only can the created actor not change them, but subsequent `changes` by the creating actor do not affect the value seen by the created actor.

Immutable declarations

I = E

Identifiers declared in this way cannot be altered except by being bound in a subsequent local declaration, `accept` statement, or actor script declaration.

Sequencing of declarations

Δ_0; Δ_1

Δ_0 is evaluated, followed by Δ_1.

Parenthesized declarations

(Δ)

Parentheses are ignored.

Actor Script Declarations

Script declaration

$(I = \Phi)$

The purpose of a script declaration is to bind an identifier I to a new actor whose initial behavior is given by Φ. See the `create` command.

Sequences of script declarations

$\Sigma_0 \ \Sigma_1$

The order of script declarations is irrelevant (except when the same identifier is used twice, in which case the compiler ought to warn the programmer). See the `create` command.

Programs

An Atolia program is an actor script declaration. The program will be started by sending an empty message to the first actor declared in the program. The program may request input from and send output to a special actor denoted by `user` in the initial environment. The actor denoted by `user` may be a terminal, a file, or an operating system.

Atolia: Sample Programs

Iterative (tail recursive) factorial subprogram:

```
(factorial = accept [ continuation n ]
             send [ continuation n 1 ] to loop)

(loop = accept [ continuation n product ]
        if or [ (lessp [ n 1 ]) (equal [ n 1 ]) ]
           then send product to continuation
           else send [ continuation
                       (minus [ n 1 ])
                       (times [ n product ]) ]
                 to loop)
```

Recursive factorial subprogram:

```
(factorial = accept [ continuation n ]
             if or [ (lessp [ n 1 ]) (equal [ n 1 ]) ]
                then send 1 to continuation
                else (create ((multiply-by-n
                                  = accept [ x ]
                                    send times [ n x ] to continuation))
                             send [ multiply-by-n (minus [ n 1 ]) ]
                               to factorial))
```

A subprogram that creates instances of queues:

```
(create-queue
  = accept [ continuation ]
    create ((queue
                = q initially [ ]
                  inside
                  accept [ c op x ]
                  if equal [ op "empty?" ]
                     then send equal [ q [ ] ] to c
                     else
                  if equal [ op "length" ]
                     then send length(q) to c
                     else
                  if equal [ op "head" ]
                     then send if equal [ q [ ] ]
                                  then "error -- empty queue has no head"
                                  else first(q)
                               to c
                     else
                  if equal [ op "enque" ]
                     then change q to append [ q [ x ] ] ;
                          send "ok" to c
                     else
                  if equal [ op "deque" ]
                     then if equal [ q [ ] ]
                             then send "error -- can't deque an empty queue"
                                    to c
                             else change q to rest(q) ;
                                  send "ok" to c
                     else
                  send "error -- unrecognized operation on queue" to c))
      send queue to continuation)
```

```
A program to calculate and print the prime numbers
using a parallel version of the Sieve of Eratosthenes:

(print-primes = accept [ go ]
                send [ print-primes "request" ] to sieve ;
                become
                  accept [ c r prime ]
                  if print (prime)
                     then send [ print-primes "request" ] to sieve
                     else dummy)

(integers = n initially 2
              inside
            accept [ c request ]
            send [ integers "reply" n ] to c ;
            change n to plus [ n 1 ])

(sieve =
  generator initially integers ;
  waiting-consumer initially [ ]
    inside
  accept [ c r prime ]
  if equal [ r "request" ]
     then change waiting-consumer to c ;
          send [ sieve "request" ] to generator
     else
  if equal [ r "reply" ]
     then send [ sieve "reply" prime ] to waiting-consumer ;
          (create ((filter =
                     waiting-consumer initially [ ] ;
                     candidate initially 0 ;
                     multiple initially prime
                       inside
                     accept [ c r n ]
                     if equal [ r "reply" ]
                        then if lessp [ multiple n ]
                                then change multiple
                                       to plus [ multiple prime ] ;
                                     send [ c r n ] to filter
                                else
                             if equal [ multiple n ]
                                then send [ filter "request" ] to generator
                                else
                             if lessp [ n multiple ]
                                then if equal [ waiting-consumer [ ] ]
                                        then change candidate to n
                                        else send [ filter "reply" n ]
                                               to waiting-consumer ;
                                             change waiting-consumer to [ ] ;
                                             send [ filter "request" ]
                                               to generator
                                else dummy
                        else
                     if equal [ r "request" ]
                        then if equal [ candidate 0 ]
                                then change waiting-consumer to c
                                else send [ filter "reply" candidate ] to c ;
                                     change candidate to 0 ;
                                     send [ filter "request" ] to generator
                        else dummy))
               send [ filter "request" ] to generator ;
               change generator to filter)
     else dummy)
```

A subprogram that acts as a stack:

```
(stack = elements initially [ ]
           inside
         accept [ continuation op x ]
         if equal [ op "push" ]
            then change elements to [ x elements ] ;
                 send "pushed" to continuation
            else
         if equal [ op "pop" ]
            then
              (if equal [ elements [ ] ]
                   then send "error -- stack empty" to continuation
                   else change elements to second(elements) ;
                        send "popped" to continuation)
            else
         if equal [ op "top" ]
            then
              (if equal [ elements [ ] ]
                   then send "error -- stack empty" to continuation
                   else send first(elements) to continuation)
            else
         if equal [ op "empty?" ]
            then send equal [ elements [ ] ] to continuation
            else
         send "error -- undefined operation on stack" to continuation)
```

The LOOP program of Chapter IV:

```
(loop = accept [ ]
        send "add1" to loop ;
        become i initially 0
                 inside
               accept [ msg ]
               if equal [ msg "add1" ]
                  then change i to plus [ i 1 ] ;
                       send "add1" to loop
                  else
               if equal [ msg "halt" ]
                  then send i to user ; become accept [ ] dummy
                  else dummy)
```

The unboundedly nondeterministic CHOOSE program of Chapter IV:

```
(choose = accept [ ]
          send "add1" to choose ;
          send "halt" to choose ;
          become i initially 0
                   inside
                 accept [ msg ]
                 if equal [ msg "add1" ]
                    then change i to plus [ i 1 ] ;
                         send "add1" to choose
                    else
                 if equal [ msg "halt" ]
                    then send i to user ; become accept [ ] dummy
                    else dummy)
```

The possibly nonterminating CHOICE-LOOP program of Chapter IV:

```
(choice-loop = accept [ ]
               send "add1" to choice-loop ;
               send "stop" to choice-loop ;
               become i initially 0 ;
                      waiting initially false
                        inside
                      accept [ msg ]
                      if waiting
                         then if equal [ msg "stop" ]
                                 then change i to plus [ i 1 ] ;
                                      change waiting to false ;
                                      send "add1" to choice-loop ;
                                      send "stop" to choice-loop
                                 else dummy
                         else if equal [ msg "add1" ]
                                 then change waiting to true
                                 else
                              if equal [ msg "stop" ]
                                 then send i to user ;
                                      become accept [ ] dummy
                                 else dummy)
```

Appendix III

Atolia: Comparison with Act1 and CSP

Atolia is a toy language designed to illustrate actors. In most respects Atolia is merely a simplified form of the experimental language Act1.[1] A multiprocessing version of a small dialect of Act1 has been implemented on the MIT Lisp Machines using the Chaosnet for interprocessor communication. Nondeterministic single processor implementations with simulated concurrency exist for Atolia on the MIT Lisp Machines and on the MIT AI Lab's PDP-10.

Act1 has a number of syntactic features not found in Atolia. Whereas in Atolia continuations must be passed as explicit message components, Act1 has conventions that allow most continuations to be suppressed. Whereas an Atolia program can create actors only through the `create` command, Act1 programs create many actors implicitly. These features of Act1 make programming easier, but the large doses of syntactic sugar obscure what is really going on in terms of actor semantics. Since illustrating actor semantics is the whole purpose of Atolia, its syntax is less refined than Act1's.

The only major semantic difference between Atolia and Act1 is that everything in Act1 is considered to be an actor. For example, the behavior of an actor in Act1 is another actor; an actor's state variables are also actors. This must not be taken too seriously because it leads to an infinite regress of message passing, as an actor consults its behavior to see what to do, and its behavior then consults *its* behavior, and so on. It is also hard to understand how a primitive serializer that has asked its behavior

[1]Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying and proving properties of guardians for distributed systems," *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979, pages 316–336.

how to act on a message it has accepted can accept the behavior's reply while remaining locked from the original message.[2]

In Atolia, however, actors correspond to network addresses identifying code segments. The behavior of an actor is not itself an actor, but is instead a mathematical function defined by the actor's code via a conventional programming language semantics. The behavior of an actor bears the same relation to the actor that the *a priori* meaning of a process bears to the process in the semantics of Communicating Sequential Processes.[3]

Actors in Atolia are similar in other ways to the processes of Communicating Sequential Processes (CSP).[4] (So are the actors of Act1, but Atolia is more like CSP than is Act1.) Like CSP processes, actors cannot access each other's local variables, and aside from actors acting as data structures there are no global variables. As with CSP processes, all interaction between actors takes place through message passing.

CSP processes whose repetitive commands have only input guards and whose alternative commands have as guards either all input guards or all boolean guards are roughly comparable to actors whose command body contains no `create` commands. Atolia has no counterpart to the automatic termination of a repetitive command with input guards, however, so an actor requires some sort of condition to become true before it proceeds to the rest of its text (using `become`).

CSP input commands must name the outputting process, while an actor can accept messages from actors it does not know about. CSP output commands cause the outputting process to wait until the target process accepts the message; an actor starts a message on its way and the actor proceeds, no permission or acknowledgement being required from the target actor. Each message sent in Atolia is eventually accepted by its target actor; a CSP output command may never finish execution because the target process never accepts the message.

CSP has nothing resembling the `create` command of Atolia. A CSP program consists of a fixed number of processes, and the intercommunication topology of those processes is static. The process

[2]The most recent version of Act1 has, in fact, backed away from some of these views.

[3]Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P deRoever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, 1979, pages 290–308.

[4]C A R Hoare, "Communicating sequential processes", *CACM* **21**, 8, August 1978, pages 666–677.

identifiers of input and output commands are constants, so that the set of processes a given process can send to or receive from is apparent from its text. Atolia, in contrast, permits actors to be created dynamically. Actor names may be passed freely in messages, and may be bound as the value of identifiers. Indeed Atolia's syntax allows arbitrary expressions to appear in the target position of **send** commands.

The fact that actors can be created does not imply that Atolia is unsuitable for implementation on a fixed network of processors. Many actors are created only to serve as explicit continuations for recursive programs; actor creation of this sort can be as inexpensive as recursive function calls in Lisp. In other instances actor creation corresponds to process creation. The questions of which actor creations should be implemented as local function calls and which should be implemented as concurrent processes can be decided by a compiler based on its knowledge of the target machine. While there may be good reasons for retaining the conventional syntactic distinctions between function calls (generating implicit continuations) and process creation, it is an achievement of the actor model that process creation and continuation creation appear the same semantically.

Appendix IV

Atolia: Formal Semantics

This appendix presents the semantics of a toy language illustrating actors, culminating in the definition of a function

$$\mathcal{P}: \mathtt{Act} \rightarrow (\mathtt{A} \rightarrow \mathtt{F})$$

giving for each program an assignment of initial behaviors to actors. This function is the starting point for the power domain semantics of Chapter IV, modified for actor creation by the changes outlined in §V.4.

The notation in this appendix is based on that of Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics.*[5] A one page summary appears at the end of this appendix. Similar notation is used by Tennent, Gordon, and Stoy (see bibliography).

[5]Chapman and Hall, London, 1976.

Atolia

(a toy language illustrating actors)

Version 0

Syntactic domains

$I \in$ `Ide`	identifiers
$B \in$ `Bas`	bases
$O \in$ `Opr`	operators
$E \in$ `Exp`	expressions
$\Gamma \in$ `Com`	commands
$\Phi \in$ `Abs`	abstractions (scripts)
$\Delta \in$ `Dec`	local declarations
$\Sigma \in$ `Act`	actor script declarations

Productions

E ::= B | OE | I | $[E_0 \cdots E_n]$
| if E_0 then E_1 else E_2 | (E)

Γ ::= dummy | change I to E | become Φ | send E_0 to E_1
| create (Σ) Γ | Γ_0; Γ_1 | if E then Γ_0 else Γ_1 | (Γ)

Φ ::= accept $[I_0 \cdots I_n]$ Γ | Δ inside Φ | if E then Φ_0 else Φ_1 | (Φ)

Δ ::= I initially E | I = E | Δ_0; Δ_1 | (Δ)

Σ ::= (I = Φ) | Σ_0 Σ_1

Value domains

$\alpha \in A = \{user\} + (\{program\} \times N) + ((A \times M^{+}) \times N)$	actors
$\gamma \in G = A \times G$	actor name generators
$\mu \in M = V$	messages
$\phi \in F = M \rightarrow G \rightarrow F \times (A \times M)^{*} \times F^{*}$	behaviors
T	truth values
$\nu \in N$	integers
R	numbers
H	characters
$B = T + N + R + H^{*}$	basic values
$\epsilon \in V = T + N + R + H^{*} + A + V^{*}$	denoted values
$\rho \in U = (Ide \rightarrow (V + \{unbound\})) \times Ide^{*}$	environments
$\chi \in X = U \rightarrow F$	behavior continuations

Semantic functions

$\mathfrak{B}$: Bas → B

$\mathfrak{O}$: Opr → V → V

$\mathfrak{E}$: Exp → U → V

$\mathfrak{N}$: Com → U → G → X → X

$\mathfrak{T}$: Com → U → G → (A x M)*

$\mathfrak{C}$: Com → U → G → F*

$\mathfrak{U}$: Com → U → G → U

$\mathfrak{G}$: Com → U → G → G

$\mathfrak{F}$: Abs → U → F

$\mathfrak{D}$: Dec → U → U

$\mathfrak{I}$: Act → U → G → U

$\mathfrak{J}$: Act → G → G

$\mathfrak{f}$: Act → U → F*

$\mathfrak{P}$: Act → A → F

$\mathcal{E}$: Exp $\to$ U $\to$ V

$\mathcal{E}[\![\mathrm{B}]\!] = \lambda\rho.\mathfrak{B}[\![\mathrm{B}]\!]$ in V

$\mathcal{E}[\![\mathrm{OE}]\!] = \lambda\rho.\mathcal{O}[\![\mathrm{O}]\!](\mathcal{E}[\![\mathrm{E}]\!]\rho)$

$\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho.(\rho \downarrow 1)[\![\mathrm{I}]\!]$

$\mathcal{E}[\![[\mathrm{E}_0 \cdots \mathrm{E}_n]]\!] = \lambda\rho.\langle \mathcal{E}[\![\mathrm{E}_0]\!]\rho, \ldots, \mathcal{E}[\![\mathrm{E}_n]\!]\rho\rangle$

$\mathcal{E}[\![$if E_0 then E_1 else $\mathrm{E}_2]\!]$

$= \lambda\rho.(\lambda\epsilon.\epsilon \sqsubseteq \mathsf{T} \to ((\epsilon \mid \mathsf{T}) \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho, \mathcal{E}[\![\mathrm{E}_2]\!]\rho), \mathit{error})$

$(\mathcal{E}[\![\mathrm{E}_0]\!]\rho)$

$\mathcal{E}[\![(\mathrm{E})]\!] = \mathcal{E}[\![\mathrm{E}]\!]$

$\mathcal{N}$: Com $\to$ U $\to$ G $\to$ X $\to$ X

$\mathcal{N}[\![$dummy$]\!] = \lambda\rho\gamma\chi.\chi$

$\mathcal{N}[\![$change I to E$]\!] = \lambda\rho\gamma\chi.\chi$

$\mathcal{N}[\![$become $\Phi]\!] = \lambda\rho\gamma\chi.(\lambda\rho'.\mathcal{T}[\![\Phi]\!]\rho)$

$\mathcal{N}[\![$send E_0 to $\mathrm{E}_1]\!] = \lambda\rho\gamma\chi.\chi$

$\mathcal{N}[\![$create (Σ) $\Gamma]\!] = \lambda\rho\gamma\chi.\mathcal{N}[\![\Gamma]\!](\mathcal{I}[\![\Sigma]\!]\rho\gamma)(\mathcal{I}[\![\Sigma]\!]\gamma)\chi$

$\mathcal{N}[\![\Gamma_0; \Gamma_1]\!] = \lambda\rho\gamma\chi.\mathcal{N}[\![\Gamma_1]\!](\mathcal{U}[\![\Gamma_0]\!]\rho\gamma)(\mathcal{G}[\![\Gamma_0]\!]\rho\gamma)(\mathcal{N}[\![\Gamma_0]\!]\rho\gamma\chi)$

$\mathcal{N}[\![$if E then Γ_0 else $\Gamma_1]\!]$

$= \lambda\rho\gamma\chi.(\lambda\epsilon.\epsilon \sqsubseteq \mathsf{T} \to ((\epsilon \mid \mathsf{T}) \to \mathcal{N}[\![\Gamma_0]\!]\rho\gamma\chi, \mathcal{N}[\![\Gamma_1]\!]\rho\gamma\chi), \mathit{error})$

$(\mathcal{E}[\![\mathrm{E}]\!]\rho)$

$\mathcal{N}[\![(\Gamma)]\!] = \mathcal{N}[\![\Gamma]\!]$

$\mathcal{T}$: Com → U → G → (A x M)*

$\mathcal{T}[\![\text{dummy}]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{T}[\![\text{change I to E}]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{T}[\![\text{become } \Phi]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{T}[\![\text{send } E_0 \text{ to } E_1]\!] = \lambda\rho\gamma.(\lambda\epsilon.\epsilon \sqsubseteq A \to \langle\langle(\epsilon \mid A), \mathcal{E}[\![E_0]\!]\rho \text{ in M}\rangle\rangle, error)(\mathcal{E}[\![E_1]\!]\rho)$

$\mathcal{T}[\![\text{create } (\Sigma)\ \Gamma]\!] = \lambda\rho\gamma.\mathcal{T}[\![\Gamma]\!](\mathcal{I}[\![\Sigma]\!]\rho\gamma)(\mathcal{J}[\![\Sigma]\!]\gamma)$

$\mathcal{T}[\![\Gamma_0;\Gamma_1]\!] = \lambda\rho\gamma.(\mathcal{T}[\![\Gamma_0]\!]\rho\gamma)\,\S\,(\mathcal{T}[\![\Gamma_1]\!](\mathcal{U}[\![\Gamma_0]\!]\rho\gamma)(\mathcal{G}[\![\Gamma_0]\!]\rho\gamma))$

$\mathcal{T}[\![\text{if E then } \Gamma_0 \text{ else } \Gamma_1]\!]$

$= \lambda\rho\gamma.(\lambda\epsilon.\epsilon \sqsubseteq T \to ((\epsilon \mid T) \to \mathcal{T}[\![\Gamma_0]\!]\rho\gamma, \mathcal{T}[\![\Gamma_1]\!]\rho\gamma), error)$

$(\mathcal{E}[\![E]\!]\rho)$

$\mathcal{T}[\![(\Gamma)]\!] = \mathcal{T}[\![\Gamma]\!]$

$\mathcal{C}$: Com → U → G → F*

$\mathcal{C}[\![\text{dummy}]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{C}[\![\text{change I to E}]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{C}[\![\text{become } \Phi]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{C}[\![\text{send } E_0 \text{ to } E_1]\!] = \lambda\rho\gamma.\langle\,\rangle$

$\mathcal{C}[\![\text{create } (\Sigma)\ \Gamma]\!] = \lambda\rho\gamma.(\mathcal{F}[\![\Sigma]\!](\mathcal{I}[\![\Sigma]\!]\rho\gamma))\,\S\,(\mathcal{C}[\![\Gamma]\!](\mathcal{I}[\![\Sigma]\!]\rho\gamma)(\mathcal{J}[\![\Sigma]\!]\gamma))$

$\mathcal{C}[\![\Gamma_0;\Gamma_1]\!] = \lambda\rho\gamma.(\mathcal{C}[\![\Gamma_0]\!]\rho\gamma)\,\S\,(\mathcal{C}[\![\Gamma_1]\!](\mathcal{U}[\![\Gamma_0]\!]\rho\gamma)(\mathcal{G}[\![\Gamma_0]\!]\rho\gamma))$

$\mathcal{C}[\![\text{if E then } \Gamma_0 \text{ else } \Gamma_1]\!]$

$= \lambda\rho\gamma.(\lambda\epsilon.\epsilon \sqsubseteq T \to ((\epsilon \mid T) \to \mathcal{C}[\![\Gamma_0]\!]\rho\gamma, \mathcal{C}[\![\Gamma_1]\!]\rho\gamma), error)$

$(\mathcal{E}[\![E]\!]\rho)$

$\mathcal{C}[\![(\Gamma)]\!] = \mathcal{C}[\![\Gamma]\!]$

$\mathcal{U}$: Com → U → G → U

$\mathcal{U}[\![\text{dummy}]\!] = \lambda\rho\gamma.\,\rho$

$\mathcal{U}[\![\text{change I to E}]\!] = \lambda\rho\gamma.\,\mathrm{I} \in (\rho \downarrow 2) \to \rho[\mathcal{E}[\![\mathrm{E}]\!]\rho/\mathrm{I}], \mathit{error}$

$\mathcal{U}[\![\text{become } \Phi]\!] = \lambda\rho\gamma.\,\rho$

$\mathcal{U}[\![\text{send } \mathrm{E}_0 \text{ to } \mathrm{E}_1]\!] = \lambda\rho\gamma.\,\rho$

$\mathcal{U}[\![\text{create } (\Sigma)\ \Gamma]\!] = \lambda\rho\gamma.\,\mathit{updates}\,\rho\,(\mathcal{U}[\![\Gamma]\!]\,(\mathcal{I}[\![\Sigma]\!]\rho\gamma)\,(\mathcal{J}[\![\Sigma]\!]\gamma))$

$\mathcal{U}[\![\Gamma_0; \Gamma_1]\!] = \lambda\rho\gamma.\,\mathcal{U}[\![\Gamma_1]\!]\,(\mathcal{U}[\![\Gamma_0]\!]\rho\gamma)\,(\mathcal{G}[\![\Gamma_0]\!]\rho\gamma)$

$\mathcal{U}[\![\text{if E then } \Gamma_0 \text{ else } \Gamma_1]\!]$

$= \lambda\rho\gamma.\,(\lambda\epsilon.\,\epsilon \in \mathsf{T} \to ((\epsilon \mid \mathsf{T}) \to \mathcal{U}[\![\Gamma_0]\!]\rho\gamma, \mathcal{U}[\![\Gamma_1]\!]\rho\gamma), \mathit{error})$

$(\mathcal{E}[\![\mathrm{E}]\!]\rho)$

$\mathcal{U}[\![(\Gamma)]\!] = \mathcal{U}[\![\Gamma]\!]$

$\mathcal{G}$: Com → U → G → G

$\mathcal{G}[\![\text{dummy}]\!] = \lambda\rho\gamma.\,\gamma$

$\mathcal{G}[\![\text{change I to E}]\!] = \lambda\rho\gamma.\,\gamma$

$\mathcal{G}[\![\text{become } \Phi]\!] = \lambda\rho\gamma.\,\gamma$

$\mathcal{G}[\![\text{send } \mathrm{E}_0 \text{ to } \mathrm{E}_1]\!] = \lambda\rho\gamma.\,\gamma$

$\mathcal{G}[\![\text{create } (\Sigma)\ \Gamma]\!] = \lambda\rho\gamma.\,\mathcal{G}[\![\Gamma]\!]\,(\mathcal{I}[\![\Sigma]\!]\rho\gamma)\,(\mathcal{J}[\![\Sigma]\!]\gamma)$

$\mathcal{G}[\![\Gamma_0; \Gamma_1]\!] = \lambda\rho\gamma.\,\mathcal{G}[\![\Gamma_1]\!]\,(\mathcal{U}[\![\Gamma_0]\!]\rho\gamma)\,(\mathcal{G}[\![\Gamma_0]\!]\rho\gamma)$

$\mathcal{G}[\![\text{if E then } \Gamma_0 \text{ else } \Gamma_1]\!]$

$= \lambda\rho\gamma.\,(\lambda\epsilon.\,\epsilon \in \mathsf{T} \to ((\epsilon \mid \mathsf{T}) \to \mathcal{G}[\![\Gamma_0]\!]\rho\gamma, \mathcal{G}[\![\Gamma_1]\!]\rho\gamma), \mathit{error})$

$(\mathcal{E}[\![\mathrm{E}]\!]\rho)$

$\mathcal{G}[\![(\Gamma)]\!] = \mathcal{G}[\![\Gamma]\!]$

$\mathcal{F}$: Abs $\rightarrow$ U $\rightarrow$ F

$\mathcal{F}[\![$accept $[\mathrm{I}_0 \cdots \mathrm{I}_n]$ $\Gamma]\!]$

$= fix(\lambda\chi.\lambda\rho.\lambda\mu\gamma.(\lambda\rho'.\langle\mathcal{N}[\![\Gamma]\!]\rho'\gamma\chi(\mathcal{U}[\![\Gamma]\!]\rho'\gamma),\ \mathcal{T}[\![\Gamma]\!]\rho'\gamma,\ \mathcal{C}[\![\Gamma]\!]\rho'\gamma\rangle)$

$\langle(divert\,\rho\,(match\,\langle \mathrm{I}_0, \ldots, \mathrm{I}_n\rangle\,(\mu \mid \mathrm{V}))) \downarrow 1, removes\,\langle \mathrm{I}_0, \ldots, \mathrm{I}_n\rangle\,(\rho \downarrow 2)\rangle)$

$\mathcal{F}[\![\Delta$ inside $\Phi]\!] = \lambda\rho.\mathcal{F}[\![\Phi]\!](\mathcal{D}[\![\Delta]\!]\rho)$

$\mathcal{F}[\![$if E then Φ_0 else $\Phi_1]\!]$

$= \lambda\rho.(\lambda\epsilon.\epsilon \sqsubseteq \mathsf{T} \rightarrow ((\epsilon \mid \mathsf{T}) \rightarrow \mathcal{F}[\![\Phi_0]\!]\rho, \mathcal{F}[\![\Phi_1]\!]\rho), error)$

$(\mathcal{E}[\![\mathrm{E}]\!]\rho)$

$\mathcal{F}[\![(\Phi)]\!] = \mathcal{F}[\![\Phi]\!]$

$\mathcal{D}$: Dec $\rightarrow$ U $\rightarrow$ U

$\mathcal{D}[\![\mathrm{I}$ initially $\mathrm{E}]\!] = \lambda\rho.\langle(\rho[\mathcal{E}[\![\mathrm{E}]\!]\rho/\mathrm{I}]) \downarrow 1, (\rho \downarrow 2) \S \langle \mathrm{I}\rangle\rangle$

$\mathcal{D}[\![\mathrm{I} = \mathrm{E}]\!] = \lambda\rho.\langle(\rho[\mathcal{E}[\![\mathrm{E}]\!]\rho/\mathrm{I}]) \downarrow 1, remove\,\mathrm{I}\,(\rho \downarrow 2)\rangle$

$\mathcal{D}[\![\Delta_0;\ \Delta_1]\!] = \lambda\rho.\mathcal{D}[\![\Delta_1]\!](\mathcal{D}[\![\Delta_0]\!]\rho)$

$\mathcal{D}[\![(\Delta)]\!] = \mathcal{D}[\![\Delta]\!]$

$\mathfrak{I}$: Act → U → G → U

$\mathfrak{I}[\![(\mathrm{I} = \Phi)]\!] = \lambda\rho\gamma\,.\,\langle(\rho[(\gamma \downarrow 1)/\mathrm{I}]) \downarrow 1, \mathit{remove}\,\mathrm{I}\,(\rho \downarrow 2)\rangle$

$\mathfrak{I}[\![\Sigma_0\ \ \Sigma_1]\!] = \lambda\rho\gamma\,.\,\mathfrak{I}[\![\Sigma_1]\!]\,(\mathfrak{I}[\![\Sigma_0]\!]\rho\gamma)\,(\mathfrak{J}[\![\Sigma_0]\!]\gamma)$

$\mathfrak{J}$: Act → G → G

$\mathfrak{J}[\![(\mathrm{I} = \Phi)]\!] = \lambda\gamma\,.\,\gamma \downarrow 2$

$\mathfrak{J}[\![\Sigma_0\ \ \Sigma_1]\!] = \lambda\rho\gamma\,.\,\mathfrak{J}[\![\Sigma_1]\!]\,(\mathfrak{J}[\![\Sigma_0]\!]\gamma)$

$\mathfrak{F}$: Act → U → F*

$\mathfrak{F}[\![(\mathrm{I} = \Phi)]\!] = \lambda\rho\,.\,\langle\mathfrak{F}[\![\Phi]\!]\,\langle\rho \downarrow 1, \langle\,\rangle\rangle\rangle$

$\mathfrak{F}[\![\Sigma_0\ \ \Sigma_1]\!] = \lambda\rho\,.\,(\mathfrak{F}[\![\Sigma_0]\!]\rho)\,\S\,(\mathfrak{F}[\![\Sigma_1]\!]\rho)$

$\mathcal{P}$: Act $\rightarrow$ A $\rightarrow$ F

$\mathcal{P}[\![\Sigma]\!] = beh\,(\mathcal{S}[\![\Sigma]\!]\,(\mathcal{I}[\![\Sigma]\!]\,\rho_{initial}\,\gamma_{initial}))$

where

$\rho_{initial} = \langle(\lambda \mathrm{I}\,.\,(\mathrm{I} = \mathtt{user}) \rightarrow user, unbound),\ \langle\,\rangle\rangle$

$\gamma_{initial} = gamma\ program$

and

beh: F* $\rightarrow$ A $\rightarrow$ F

is defined by

$beh\,\phi^*\,user = fix(\lambda\phi\,.\,\lambda\mu\gamma\,.\,\langle\phi, \langle\,\rangle, \langle\,\rangle\rangle)$

$beh\,\phi^*\,\langle program, \nu\rangle = \phi^* \downarrow (\nu + 1)$

$beh\,\phi^*\,\langle\langle\alpha, \mu^*\rangle, \nu\rangle = (((behav\,\alpha\,(beh\,\phi^*\alpha)\,(droplast\,\mu^*))$

$(last\,\mu^*)\,(gamma\,\langle\alpha, \mu^*\rangle)) \downarrow 3) \downarrow (\nu + 1)$

$last\,\mu^* = \mu^* \downarrow \#\mu^*$

$droplast\,\mu^*$ is the sequence μ_0^* such that $\mu_0^* \S \langle\mu^* \downarrow \#\mu^*\rangle = \mu^*$.

Auxiliary functions

$\rho[\epsilon/\mathrm{I}] = \langle(\lambda \mathrm{I}' = \mathrm{I} \rightarrow \epsilon, (\rho \downarrow 1)\,[\![\mathrm{I}']\!]), \rho \downarrow 2\rangle$

$arid = \langle(\lambda \mathrm{I}\,.\,unbound), \langle\,\rangle\rangle$

$divert\colon \mathsf{U} \rightarrow \mathsf{U}$

$divert\rho_0\rho_1 = \langle(\lambda \mathrm{I}\,.\,(\rho_1 \downarrow 1)\,[\![\mathrm{I}]\!] \neq unbound \rightarrow (\rho_1 \downarrow 1)\,[\![\mathrm{I}]\!], (\rho_0 \downarrow 1)\,[\![\mathrm{I}]\!]),\ (\rho_0 \downarrow 2)\,\S\,(\rho_1 \downarrow 2)\rangle$

$remove\colon \mathtt{Ide} \rightarrow \mathtt{Ide}^* \rightarrow \mathtt{Ide}^*$

$remove\,\mathrm{I}\,x = (0 = \#x) \rightarrow \langle\,\rangle, (\mathrm{I} = (x \downarrow 1) \rightarrow remove\,\mathrm{I}\,(x \dagger 1),$

$\langle(x \downarrow 1)\rangle\,\S\,remove\,\mathrm{I}\,(x \dagger 1))$

$removes\colon \mathtt{Ide}^* \rightarrow \mathtt{Ide}^* \rightarrow \mathtt{Ide}^*$

$removes\,\mathrm{I}^*\,x = (0 = \#\mathrm{I}^*) \rightarrow x, removes\,(\mathrm{I}^* \dagger 1)$

$(remove\,(\mathrm{I}^* \downarrow 1)\,x)$

$updates\colon \mathsf{U} \rightarrow \mathsf{U} \rightarrow \mathsf{U}$

$updates\,\rho_0\rho_1 = (\lambda \mathrm{I}^*\,.\,(0 = \#(\rho_1 \downarrow 2)) \rightarrow \rho_0, (\lambda \mathrm{I}\,.\,updates\,(\rho_0[(\rho_1 \downarrow 1)\,[\![\mathrm{I}]\!]/\mathrm{I}])$

$\langle(\rho_1 \downarrow 1), (\mathrm{I}^* \dagger 1)\rangle)$

$(\mathrm{I}^* \downarrow 1))$

$(\rho_1 \downarrow 2)$

$match$: $\mathsf{Ide}^* \to \mathsf{V} \to \mathsf{U}$

$match\,\mathrm{I}^*\,\epsilon$

$= (0 = \sharp\mathrm{I}^*) \to arid,$

$\epsilon \sqsubseteq \mathsf{V}^* \to (0 = \sharp(\epsilon \mid \mathsf{V}^*) \to divert\,(arid[\langle\,\rangle/\mathrm{I}])\,(match\,(\mathrm{I}^* \dagger 1)\,\epsilon),$

$divert\,(arid[(\epsilon \downarrow 1)/\mathrm{I}])\,(match\,(\mathrm{I}^* \dagger 1)\,(\epsilon \dagger 1))),$

$divert\,(arid[\epsilon/\mathrm{I}])\,(match\,(\mathrm{I}^* \dagger 1)\,\epsilon)$

$gamma$: $(\{program\} + (\mathsf{A} \times \mathsf{M}^+)) \to \mathsf{G}$

$gamma\,x = (fix(\lambda f\,.\,\lambda x\nu\,.\,\langle\langle x, \nu\rangle, f\,x\,(\nu + 1)\rangle\rangle)$

$x\,0$

$behav$: $\mathsf{A} \to \mathsf{F} \to \mathsf{M}^* \to \mathsf{F}$

$behav\,a\phi\mu^*$

$= (fix(\lambda f\,.\,\lambda a\phi\mu_0^*\mu_1^*\,.\,(0 = \sharp\mu_0^*) \to \phi, f\,a$

$((\phi\,(\mu_0^* \downarrow 1)\,(gamma\,\langle a, \mu_1^* \S \langle\mu_0^* \downarrow 1\rangle\rangle)) \downarrow 1)$

$(\mu_0^* \dagger 1)$

$(\mu_1^* \S \langle\mu_0^* \downarrow 1\rangle)))$

$a\phi\mu^*\langle\,\rangle$

Notation

All domains in this appendix are complete lattices.

The separated sum of lattices $D_0, \ldots, D_n$ is defined in Milne and Strachey, and is written $D = D_0 + \cdots + D_n$. If x belongs to the sum D, then $x \in D_i$ tells whether x is in the summand D_i. $x \mid D_i$ is the projection of x to D_i, while y in D indicates the injection of y into D for y a member of a summand of D. While some of the semantic equations may omit some injections and projections, injections and projections into and from the domains A, G, and M will always be given explicitly. These domains must be treated with care because it is easy to confuse some of their elements with elements of V. For example, an element $\epsilon \in$ V can never be an element of M although ϵ in M is always an element of M. Similarly no element of A is an element of V*.

The product of lattices is written $D_0 \times \cdots \times D_n$. Elements of the product are written $\langle x_0, \ldots, x_n \rangle$, and the projections are indicated by $\langle x_0, \ldots, x_n \rangle \downarrow i + 1 = x_i$.

D^* is the lattice of finite sequences from D, including the empty sequence $\langle\ \rangle$. If δ is a metavariable used to range over the domain D, then δ^* indicates an arbitrary element of D^*. The length of a sequence δ^* is indicated by $\#\delta^*$, so that $\#\langle\ \rangle = 0$ and $\#\langle \delta_0, \ldots, \delta_n \rangle = n + 1$ when $n \geq 0$. Projections are indicated by $\langle \delta_0, \ldots, \delta_n \rangle \downarrow i + 1 = \delta_i$. $\delta_0^* \S \delta_1^*$ is the concatenation of δ_0^* and δ_1^*. $\delta^* \dagger n$ indicates the finite sequence obtained by dropping the first n elements of the sequence δ^*. When δ^* is a sequence, $x \in \delta^*$ tells whether there exists an integer i such that $\delta^* \downarrow i = x$.

$D_0 \rightarrow D_1$ is the lattice of continuous functions from D_0 to D_1. Unlike sums and products, function lattices are always formed from exactly two domains. $D_0 \rightarrow D_1 \rightarrow D_2$ is taken to mean $D_0 \rightarrow (D_1 \rightarrow D_2)$.

Function application is indicated by juxtaposition, associating to the left unless parentheses instruct otherwise. Lambda abstraction is written $\lambda x \,.\, y$.

fix is the usual fixed point operator.

$x \rightarrow y_0, y_1$ is y_0 if x is true, y_1 if x is false, undefined if x is undefined, and *error* if $x =$ *error*. Each domain is assumed to have a special element *error* that is to be preserved under all the semantic equations, though the special tests for *error* have been left out of the equations in the interest of informal clarity.

References

R J Back, "Semantics of unbounded nondeterminism", Mathematisch Centrum Report IW 135/80, April 1980.

Henry Baker, "Actor systems for real-time computation", MIT LCS Technical Report 197, March 1978.

Valdis Berzins, "An independence result for actor laws", MIT LCS Computation Structures Group Note 34, December 1977.

Stephen L Bloom, "Varieties of ordered algebras", *J Computer and System Sciences* **13**, 2, October 1976, pages 200–212.

J M Cadiou and J J Levy, "Mechanizable proofs about parallel processes", Proceedings 14th Annual Symposium on Switching and Automata Theory, October 1973, pages 34–48.

Edsger Dijkstra, *A Discipline of Programming*, Prentice Hall, 1976.

Nissim Francez, C A R Hoare, Daniel J Lehmann, and Willem P de Roever, "Semantics of nondeterminism, concurrency, and communication", *J Computer and System Sciences* **19**, December 1979, pages 290–308.

W R Frantz, "Simula language summary", presented at the ACM SIGPLAN History of Programming Languages Conference, Los Angeles CA, June 1978, preprint in *SIGPLAN Notices* **13**, 8, August 1978, pages 243–244.

Daniel P Friedman and David S Wise, "Aspects of applicative programming for parallel processing", *IEEE Transactions on Computers* **C-27**, 4, April 1978, pages 289–296.

Michael J C Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

Irene Greif, "Semantics of communicating parallel processes", MIT Project MAC Technical Report 154, September 1975.

M C B Hennessy and G D Plotkin, "Full abstraction for a simple parallel programming language", FOCS-79, Springer-Verlag Lecture Notes in Computer Science 74, 1979.

Carl Hewitt, "Planner: a language for manipulating models and proving theorems in a robot", *Proceedings of the First International Joint Conference on Artificial Intelligence*, Washington DC, 1969.

Carl Hewitt, "Viewing control structures as patterns of passing messages", *Artificial Intelligence* **8**, 1977, pages 323-363. Also in Winston and Brown [ed], Artificial Intelligence: an MIT Perspective, MIT Press, 1979.

Carl Hewitt, Giuseppe Attardi, and Henry Lieberman, "Specifying and proving properties of guardians for distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Notes in Computer Science 70, 1979.

Carl Hewitt and Henry Baker, "Actors and continuous functionals", IFIP Working Conference on Formal Description of Programming Concepts, St Andrews, New Brunswick, Canada, August 1977, 16.1–16.21.

Carl Hewitt and Henry Baker, "Laws for communicating parallel processes", IFIP-77, Toronto, August 1977, pages 987–992.

C A R Hoare, "Communicating sequential processes", *CACM* **21**, 8, August 1978, pages 666–677.

Micha Hofri, "Disk scheduling: FCFS vs. SSTF revisited", *CACM* **23**, 11, November 1980, pages 645–653.

Daniel H H Ingals, "The Smalltalk-76 programming system: design and implementation", *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson AZ, January 1978, pages 9–16.

Gilles Kahn and David McQueen, "Coroutines and networks of parallel processes", IFIP-77, Montreal, August 1977, pages 993–998.

Kenneth M Kahn, "An actor-based animation language", *Creative Computing* **6**, 11, November 1980, pages 75–84.

Bill Kornfeld, "ETHER—a parallel problem solving system", IJCAI-79, pages 490–492.

Paul Roman Kosinski, "Denotational semantics of determinate and non-determinate data flow programs", MIT LCS Technical Report 220, May 1979.

Richard H Lathwell, "Some implications of APL order-of-execution rules", APL79, *APL Quote Quad* **9**, 4-Part 1, June 1979, pages 329–332.

Daniel Lehmann, "On the algebra of order", *J Computer and System Sciences* **21**, 1, August 1980, pages 1–23.

Nancy A Lynch and Michael J Fischer, "On describing the behavior and implementation of distributed systems", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.

Saunders MacLane, *Categories for the Working Mathematician*, Springer-Verlag, New York, 1971.

G Markowsky and B K Rosen, "Bases for chain-complete posets", *IBM J Research and Development* **20**, 2, March 1976, pages 138–147.

George Milne and Robin Milner, "Concurrent processes and their syntax", *JACM* **26**, 2, April 1979, pages 302–321.

Robert Milne and Christopher Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, London, 1976.

Ted Nelson [editor], "Symposium on actor languages", *Creative Computing* **6**, 10, October 1980, pages 61–86, continued in *Creative Computing* **6**, 11, November 1980, pages 74–94.

Mogens Nielsen, Gordon Plotkin, and Glynn Winskel, "Petri nets, event structures and domains", in *Semantics of Concurrent Commputation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979, pages 266–284.

Kristen Nygaard and Ole-Johan Dahl, "The development of the Simula language", presented at the ACM SIGPLAN History of Programming Languages Conference, Los Angeles CA, June 1978, preprint in *SIGPLAN Notices* **13**, 8, August 1978, pages 245–272.

David Park, "On the semantics of fair parallelism", University of Warwick Theory of Computation Report 31, October 1979.

G D Plotkin, "A powerdomain construction", *SIAM J Computing* **5**, 3, September 1976, pages 452–487.

Jerald S Schwarz, "Denotational semantics of parallelism", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.

Dana Scott, "What is Denotational Semantics?", MIT Laboratory for Computer Science Distinguished Lecture Series, 17 April 1980.

Dana Scott, "Data types as lattices", *SIAM J Computing* **5**, 3, September 1976, pages 522–587.

Raymond Smullyan, *First Order Logic*, Springer-Verlag, New York, 1968.

M B Smyth, "Power domains", *J Computer and System Sciences* **16**, 1978, pages 23–36.

M B Smyth and G D Plotkin, "The category-theoretic solution of recursive domain equations", Proceedings 18th Annual IEEE Symposium on Foundations of Computer Science, 1977, pages 13–17.

Guy Lewis Steele Jr and Gerald Jay Sussman, "Scheme: an interpreter for extended lambda calculus", MIT AI Memo 349, December 1975.

Guy Lewis Steele Jr and Gerald Jay Sussman, "The revised report on Scheme: a dialect of Lisp", MIT Artificial Intelligence Memo 452, January 1978.

Joseph E Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*, MIT Press, Cambridge MA, 1977.

R D Tennent, "The denotational semantics of programming languages", *CACM* **19**, 8, August 1976, pages 437–453.

William W Wadge, "An extensional treatment of dataflow deadlock", in *Semantics of Concurrent Computation*, Springer-Verlag Lecture Notes in Computer Science 70, 1979.