Full Abstraction and Fixed-Point Principles for Indeterminate Computation

James R. Russell

Ph.D Thesis

TR 90-1120
April 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

FULL ABSTRACTION AND FIXED-POINT PRINCIPLES FOR INDETERMINATE COMPUTATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

James Robert Russell

May 1990

FULL ABSTRACTION AND FIXED-POINT PRINCIPLES FOR INDETERMINATE COMPUTATION

James Robert Russell, Ph.D.

Cornell University 1990

Recently, there has been much interest in the problem of finding semantic models for various kinds of concurrent systems. A common property of concurrent systems is indeterminate behavior, either because of unpredictable interactions between processes, or because of abstractions removing the temporal details of the interaction. As a result, the study of the semantics of indeterminacy is necessary and important to the understanding of concurrency. The goal of any semantic model is that it capture our intuitions about the operational behavior of the underlying system and aid in our reasoning about it. Two properties of semantic models that we find useful in achieving this goal are full abstraction and fixed-point principles. In this thesis we investigate the problem of finding semantic descriptions with these properties for indeterminate systems.

We begin by looking at a simple imperative language containing unbounded indeterminacy, based on one studied by Apt and Plotkin. We use category-

theoretic techniques to develop a fixed-point semantics that, while not fully abstract, reduces to a fully abstract semantics via a simple abstraction functor.

We then concentrate on the more general setting of dataflow networks and the hierarchy of indeterminate merge primitives. We show that the straightforward generalization of Kahn's semantics based on the input-output relation fails to be compositional for any class of indeterminate dataflow networks. We then extend previous results and show that a semantics based on traces is fully abstract for all indeterminate and determinate dataflow networks, thereby providing a model considerably more general than Kahn's.

This generalization has the drawback that it does not have a simple fixed-point principle. We proceed to study a class of networks that models purely internal indeterminacy, called *oraclizable* networks, and show that for this class a generalization of Kahn's semantics to sets of functions is both fully abstract and has a natural fixed-point principle. We also show that the oraclizable networks are in fact universal for this representation. Finally, we use this representation to compare the class of oraclizable networks to other classes, and discover new relations among the classes.

Biographical Sketch

James Robert Russell was born in Pittsburgh, Pennsylvania on June 25, 1964. In 1969 he moved to Wilton, Connecticut, where he attended school for thirteen years, graduating from Wilton High School in 1982. This period of his educational background was largely uneventful, except that at some time during his high school years he first met Susan Blodgett. He went on to attend the Massachusetts Institute of Technology, where he pursued a double major in Mathematics and Computer Science. In 1986 he was awarded two bachelors degrees by MIT, graduating 1111th out of a class of 1127. This was not his class rank, but rather his relative position in line to receive his degrees, which would not have been so memorable were it not for the fact that it was an outdoor ceremony held in the pouring rain. In May 1989, he was awarded an M.S. by Cornell University. In May 1990 he will be awarded a PhD in Computer Science by Cornell. In June 1990, many years and five diplomas after first meeting her, he will marry Susan Blodgett.

To my family,

and to Sue.

Acknowledgements

First, I would like to thank Prakash Panangaden for being a great advisor. His advice on subjects academic, professional, and beyond has been invaluable to me throughout my time at Cornell. I would also like to thank Robert Constable and Anil Nerode for serving on my committee, and for their useful suggestions. On subjects semantical and otherwise, I have benefited from my interactions with Gene Stark, Bard Bloom, Dexter Kozen, and Rich Zippel.

I would like to thank all my friends in Computer Science, who from cooking dinner together to playing hockey or bridge, have made my years in Ithaca much more enjoyable.

My family has provided constant support and encouragement throughout my academic career, and I thank them all very much. Finally, for her support and patience, I thank Sue, whom I owe far too much to express in words.

Table of Contents

List of Figures

Chapter 1

Introduction

Recently, there has been much interest in the problem of finding semantic models for various kinds of concurrent systems. A common property of concurrent systems is a certain amount of indeterminate behavior, either because of unpredictable interactions between processes, or because we wish to abstract away the temporal details of the interaction. As a result, the study of the semantics of indeterminacy is necessary and important to the understanding of concurrency. The goal of any semantic model is that it capture our intuitions about the operational behavior of the underlying system and aid in our reasoning about the system. Two properties of semantic models that we find useful in achieving this goal are full abstraction and fixed-point principles. In this thesis we will investigate the problem of finding semantic descriptions with these properties for indeterminate systems.

1.1 Properties of a Model

We would like any semantic description of a programming language or system to be faithful to our concrete operational understanding of the behavior of the system [note: henceforth in this section we will use terminology appropriate for describing a sequential programming language, though the settings to which this discussion apply may be much more general]. In order to achieve this, we first must have a way of describing the observable operational behavior. Formally, we assume we have some *operational semantics* $Op[\![\;]\!]$, such that $Op[\![A]\!]$ describes the observable behavior of the given program A. What may be observed and what constitutes an observation varies according to the setting. In some cases it may be that only full programs may be observed, and hence $Op[\![\;]\!]$ is only defined for that subset of all phrases of the language that constitute full programs.

One of the advantages of a denotational semantic model is that it typically can be used to reason about arbitrary program phrases. We wish to define an equivalence on our language, based on the operational semantics, that determines when two programs or program phrases are operationally identical. We do this by employing the notion of program contexts. A program context $\mathcal{C}[\,]$ is just a program with "holes" into which a phrase A may be put, such that the completed program is observable (i.e. is in the domain of $Op[\![\;]\!]$). For example, in the typed lambda calculus, $\mathcal{C}[\,]$ is typically required to yield a closed term of ground type. We then call two program phrases equivalent if they can be substituted freely for one another in any program context without changing the observable behavior of the completed context. In other words, we define $A \equiv_{op} B$ if we have $Op[\![\mathcal{C}[A]]\!] = Op[\![\mathcal{C}[B]]\!]$ for all program contexts $\mathcal{C}[\,]$. Note that A and B may themselves be observable, but that $Op[\![A]\!] = Op[\![B]\!]$ doesn't necessarily imply $A \equiv_{op} B$.

Two closely related properties that we require of almost any semantic model are *compositionality* and *adequacy*. Compositionality states that the denotation of a composite program or process can be determined from the denotations of its components. Thus, if $\mathcal{D}[\![\,]\!]$ is our semantic mapping, then a compositional semantics has the property that if $\mathcal{D}[\![A]\!] = \mathcal{D}[\![B]\!]$ then $\mathcal{D}[\![\mathcal{C}[A]]\!] = \mathcal{D}[\![\mathcal{C}[B]]\!]$ for all contexts $\mathcal{C}[\,]$. That is, if two phrases are identified in the model, then any composite programs built using them will have identical denotations in the semantic model.

The property of adequacy states something similar – that if two phrases are identified in the model, then they are operationally equivalent and any composite programs built using them will have identical operational semantics. That is, if $\mathcal{D}[\![A]\!] = \mathcal{D}[\![B]\!]$ then $Op[\![\mathcal{C}[A]]\!] = Op[\![\mathcal{C}[B]]\!]$ for all contexts $\mathcal{C}[\,]$. This is a very reasonable property to require; it simply ensures that if two processes can be observed to be different in some operational context, then they are distinguished in the denotational model.

Adequacy directly relates the denotational and operational semantics, but it allows a model to be over-distinguishing, perhaps making denotational distinctions between programs that in fact behave identically. This may be the case when a model fails to abstract away enough information irrelevant to observability. The stronger property of *full abstraction* was introduced by Milner [Mil75] to describe the situation where this does not occur. A semantic model is fully abstract when the converse of adequacy holds as well, and it contains the minimal amount of information necessary to distinguish between observably different processes. That is, $\mathcal{D}[\![A]\!] = \mathcal{D}[\![B]\!]$ if and only if $Op[\![\mathcal{C}[A]]\!] = Op[\![\mathcal{C}[B]]\!]$ for all contexts $\mathcal{C}[\,]$. It is clear that a fully abstract semantics, while much harder to achieve, exactly captures our operational understanding of the language.

A final, somewhat different property we desire of a semantic model is that it have a fixed-point principle. Most useful languages contain the ability to loop or iterate, and it is desirable to be able to describe the meaning of such constructs as least fixed points of some recursive expression. Having this property allows us to build up such descriptions as a limit of a sequence of approximations, and allows us to reason about the constructs using an inductive definition.

1.2 Indeterminacy

A great deal of work has been done on the semantics of determinate languages. As the properties of determinate languages became well understood, people naturally began to look at more complex settings, such as concurrency and indeterminacy. In this thesis we concentrate on indeterminate settings – particularly unboundedly indeterminate settings. Unbounded indeterminacy may arise naturally in many situations, especially those involving fairness constraints (as with scheduling or resource allocation), and also those involving unknown external input or unpredictable delays in communication. Real systems such as operating systems must be able to deal with the presence of such indeterminacy, hence a greater understanding of it via well constructed semantic descriptions is important and useful.

In this thesis we study two different abstract indeterminate settings. The first is a simple imperative language with an atomic unbounded assignment primitive. The second is dataflow networks, which is a slightly more realistic setting involving communicating concurrent processes.

1.3 Overview

In Chapter 2 we study a simple imperative language containing an unbounded random assignment primitive, and use category-theoretic techniques to develop a natural continuous least fixed-point semantics for it that abstracts to a fully abstract semantics. We also show that this result is optimal in a certain sense, given results of Apt and Plotkin. In Chapter 3 we introduce the different setting of dataflow networks, which is the focus of our study for the remainder of the thesis. We present the basic definitions, describe the work of Kahn on determinate dataflow networks, and describe the various indeterminate primitives for this setting. In Chapter 4 we present the result that the straightforward generalization of Kahn's semantics fails to be compositional for even the weakest form of indeterminacy. We go on to extend previous results and show that a semantics based on traces is fully abstract for all indeterminate and determinate dataflow networks, thereby providing a model considerably more general than Kahn's. However, the trace model, while general enough to represent the full range of indeterminacy, does not have a convenient fixed-point property. In Chapter 5 we consider *oraclizable* networks, a class of networks that models the situation where indeterminacy results from imperfect knowledge of the inputs to a system. We develop a semantics for this class that is a natural generalization of Kahn's, and we prove that it is both fully abstract and has a simple fixed-point principle. In Chapter 6 we use the results of both the previous chapters to show that the class of oraclizable networks is in fact universal for the semantic model developed in Chapter 5. We also use these results to relate the oraclizable networks to other classes of indeterminate networks, and we discover several new relations among these classes. In Chapter 7 we describe and discuss related work, and in Chapter 8 we present conclusions and future directions.

Chapter 2

A Category-Theoretic Semantics for Unbounded Indeterminacy

2.1 Introduction

In this chapter we present a self-contained study of a simple imperative language containing unbounded indeterminacy and a category-theoretic semantic description of it. In a recent paper, Apt and Plotkin [AP86] showed that it is impossible to have a continuous, least fixed-point, fully abstract semantics using domain theory for such a language. This proof does not depend on the details of any particular powerdomain construction. We show that by using category-theoretic methods developed by Lehmann [Leh76], one can get a "continuous" semantics in which the meanings of while loops are given by colimits of ω-diagrams. The semantics that we provide is adequate, though not fully abstract. However it collapses, via an abstraction function, to a semantics that coincides with operational equality and *is* fully abstract. Apt and Plotkin [AP86] give a semantics for this language that is fully abstract, but it is not continuous. The failure of continuity in our case is isolated to the abstraction function.

Lehmann's category-theoretic approach to powerdomains was based on the idea that morphisms in a category could convey a more precise notion of approximation than partial orders could. He developed category-theoretic generalizations of many standard domain-theoretic concepts including the Smyth powerdomain [Smy78]. He did not use these constructions for defining the semantics of any languages. Several years later, Samson Abramsky [Abr83] sketched a semantics for an applicative language with unbounded indeterminacy and investigated mathematical properties of the resulting powerdomains. The language that he studied had its operational semantics defined via a term rewriting system. The study in this chapter is in the context of an imperative language, i.e. one with an updatable store. This work gives a more detailed study of the relationship between the operational semantics and the denotational semantics. Abramsky's study of this relationship is done by defining an appropriate operational preorder on computation sequences, along lines suggested by Boudol [Bou80], and relating this operational preorder to the categorical approximation. Boudol's analysis of the operational semantics of an applicative language could probably be mimicked in the setting of this chapter, but the treatment of the operational semantics given should be more perspicuous to most readers. This work clearly owes much to Abramsky's treatment of the subject though the details are developed differently.

The rest of this chapter is organized as follows: Section 2.2 describes the language and its operational semantics. In Section 2.3 we describe categorical powerdomains; this material is essentially a summary of the relevant parts of Lehmann's thesis [Leh76]. Sections 2.4 and 2.5 describe the semantics and provide some examples. In Section 2.6 we establish the relationship between the operational semantics and the denotational semantics.

2.2 Operational Semantics of the Language

We describe a simple language for indeterminacy based on the one presented in Apt and Plotkin [AP86]. Our language is a simplified version of that one, the state consisting of the value of a single integer variable, usually called x. It should be clear that the treatment extends readily to any "flat" domain of states.

The Atomic Commands are the ones that change the state, i.e. set the value of x.

$$A \ ::= \ x := n \mid x := x - 1 \mid x := x + 1 \mid x :=?$$

The intended meaning of $x :=?$ is indeterminate assignment to x of any value from the underlying domain (which in this case is the integers). The details of the Boolean expressions are not important, since they are intended to be side-effect free. We assume at least the ability to detect when x is 0.

$$B \ ::= \ x = 0 \mid x \neq 0 | \cdots$$

Finally, we have the following Commands:

$$S \ ::= \ A \mid \mathsf{skip} \mid S_1; S_2 \mid \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}$$

We give the operational semantics via a one-step transition function **Com** $\times$ **States** $\rightarrow$ **Com** $\times$ **States** $\cup$ **States**. We assume the existence of a boolean evaluation function $\mathcal{B}[\![\cdot]\!] : \mathbf{Bexp} \rightarrow (\mathbf{States} \rightarrow \{\mathsf{true}, \mathsf{false}\})$; we explicitly assume that the evaluation of boolean expressions terminates. In what follows we use σ to range over **States**.

$\langle x := n,\, \sigma \rangle \ \rightarrow n$

$\langle x := x - 1,\, \sigma \rangle \ \rightarrow \sigma - 1$

$\langle x := x + 1,\, \sigma \rangle \ \rightarrow \sigma + 1$

$\langle x :=?,\, \sigma \rangle \ \rightarrow n \ \forall n \in D$

$\langle \mathsf{skip},\, \sigma \rangle \ \rightarrow \sigma$

$\langle S_1; S_2,\, \sigma \rangle \ \rightarrow \langle S_1'; S_2,\, \sigma' \rangle$ if $\langle S_1,\, \sigma \rangle \ \rightarrow \langle S_1',\, \sigma' \rangle$

$\langle S_1; S_2,\, \sigma \rangle \ \rightarrow \langle S_2,\, \sigma' \rangle$ if $\langle S_1,\, \sigma \rangle \ \rightarrow \sigma'$

$\langle \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi},\, \sigma \rangle \ \rightarrow \langle S_1,\, \sigma \rangle$ if $\mathcal{B}[\![B]\!]\sigma = \mathsf{true}$

$\langle \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi},\, \sigma \rangle \ \rightarrow \langle S_2,\, \sigma \rangle$ if $\mathcal{B}[\![B]\!]\sigma = \mathsf{false}$

$\langle \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od},\, \sigma \rangle \ \rightarrow \langle S; \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od},\, \sigma \rangle$ if $\mathcal{B}[\![B]\!]\sigma = \mathsf{true}$

$\langle \mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od},\, \sigma \rangle \ \rightarrow \sigma$ if $\mathcal{B}[\![B]\!]\sigma = \mathsf{false}$

We define the notation $\langle S,\, \sigma \rangle \uparrow$ to mean that there exists an infinite sequence of transitions

$$\langle S,\, \sigma \rangle = \langle S_0,\, \sigma_0 \rangle \rightarrow \langle S_1,\, \sigma_1 \rangle \rightarrow \langle S_2,\, \sigma_2 \rangle \rightarrow \cdots,$$

and we define the operational meaning function $Op[\![\,]\!]$ by

$$Op[\![S]\!]\sigma \overset{\text{def}}{=} \{\sigma' | \langle S,\, \sigma \rangle \rightarrow^* \sigma'\} \cup \{\bot | \langle S,\, \sigma \rangle \uparrow\}.$$

2.3 Categorical Powerdomains

Powerdomains were originally introduced as the domain-theoretic analogues of powersets and were intended to be used for semantic treatments of indeterminacy [Plo76,Smy78]. The approach was generalized to categories by Lehmann in [Leh76]. Lehmann's approach was to use categories as the semantic spaces rather than domains. His idea was that a 'more detailed' notion of approximation between elements can be expressed by using morphisms between objects in a category than by using partial orders. Dually, one may view traditional domain theory as a special case of a category-theoretic approach in which the homsets are at most singletons.

Abramsky [Abr83] used Lehmann's construction to model unbounded indeterminacy. A categorical approach, but with a different construction for the powerdomain, was also used by Panangaden [Pan85] to model dataflow networks with fair merge. Recently it has been realized that one can use categories to model lambda calculi [Coq88].

Though Lehmann's original construction was for domains that were generalized to categories, in this work we assume that the underlying domain is a partial order. As already noted, a poset can be viewed as a category in which the arrows are unique; we will often adopt this view of our domain when discussing the powerdomain construction.

Given a domain E containing the least element $\perp$ the construction of what we will call the *categorical powerdomain* of E, $CP(E)$, is straightforward. The objects of the category are taken to be multisets – sets with repetitions – of the elements of E. Multisets are represented as tagged sets, i.e. each element of a multiset M is written e_α, where $e \in E$, and α is chosen from some (uncountable) set Γ of tags so that no two elements of M have the same tag. Arrows $G : A \longrightarrow B$ are defined such that for each $b_\beta \in B$, G uniquely associates with b_β an $a_\alpha \in A$ and an arrow $a \longrightarrow b$ of E. In our case where E is a poset, arrows of E are always unique and G need only associate with each $b_\beta \in B$ an $a_\alpha \in A$ with $a \sqsubseteq b$. Hence, we will use the following definition of an arrow G:

$$\begin{aligned} G \subseteq A \times B \text{ such that } \quad & i) \ \langle a_\alpha, b_\beta \rangle \in G \Rightarrow a \sqsubseteq b \ \ \& \\ & ii) \ \forall b_\beta \in B \exists! a_\alpha \in A . \langle a, b \rangle \in G. \end{aligned}$$

Composition of arrows and identity arrows in $CP(E)$ are straightforward.

Note that two multisets with the same number of copies of the same element, but possibly different tags, are isomorphic as objects of $CP(E)$. In general, it is important to specify the way in which functors or arrows act on the differently tagged elements, and the details of these constructions can be quite subtle. However, the constructions described in the rest of this chapter are sufficiently clear that explicitly stating the tagging is more cumbersome than illuminating, and hence the tagging details are omitted.

The categorical analog of a continuous function is a functor that preserves colimits of ω-diagrams. The definition of f a functor automatically assures the analog of monotonicity – i.e. if there is an arrow $a \xrightarrow{r} b$, then there is an arrow $f(a) \xrightarrow{f(r)} f(b)$.

We now present two theorems about $CP(E)$ originally due to Lehmann. These establish that $CP(E)$ has the properties that one associates with a complete partial order.

Theorem 2.1. $\{\perp\}$ is the initial object.

Proof: For all $e \in E$ we have $\perp \sqsubseteq e$, hence for any object B of $CP(E)$ there is a unique arrow $\{\perp\} \longrightarrow B$ given by $G = \{\langle \perp, b\rangle | b \in B\}$. ∎

Theorem 2.2. All ω-diagrams have colimits

Proof (sketch): Consider a diagram $X_0 \xrightarrow{G_0} X_1 \xrightarrow{G_1} X_2 \xrightarrow{G_2} \cdots$.

Let S be the collection of all chains $Q = \{q_0, q_1, \ldots\}$ s.t. for all i we have $q_i \in X_i$ and $\langle q_i, q_{i+1}\rangle \in G_i$. We define the colimiting object X^* by $X^* = \text{colim}(X_i) = \biguplus_{Q \in S} \sqcup Q$. The colimiting arrows $X_i \xrightarrow{G_i^*} X^*$ are given by $G_i^* = \biguplus_{Q \in S}\{\langle x, \sqcup Q\rangle | x \in X_i \cap Q\}$. We omit the proof that this is the colimit. ∎

Note that in the above (and in the sequel) we use the 'tagged union' symbol $\uplus$ for unions that yield multisets. The intention is that the union guarantees that each element of the result has a unique tag, and hence multiple copies of the same element will not be identified.

2.4 Semantics of the Language

In this section we use the categorical powerdomain construction defined above to give a denotational semantics to our language. First, we need to define our domains. Our base domain D will be an unrelated set of states (which in this case is the integers ω). However, since non-termination is a possibility in the language, we must consider $CP(D_\perp)$, the categorical powerdomain of D with

$\perp$ added. We write it in this way to emphasize the special treatment that $\perp$ requires. Our semantics will then be a function **Com** $\rightarrow (D \rightarrow CP(D_\perp))$, i.e. the meaning of commands will be given as ω-colimit preserving functors from D to $CP(D_\perp)$.

Before we can give the semantics, there are three auxiliary functors we must define:

Singleton $\{\cdot\}$: This takes an element d of D to the object $\{d\}$ (the singleton multiset containing one copy of d) of $CP(D_\perp)$. It takes an arrow $a \longrightarrow b$ in D to the arrow $\{a\} \xrightarrow{G} \{b\}$ given by $G = \{\langle a, b\rangle\}$. It is easily seen that this is ω-colimit preserving.

Lifting $(\cdot)^\dagger$: This is a functor between the functor categories $(D \rightarrow CP(D_\perp))$ and $(CP(D_\perp) \rightarrow CP(D_\perp))$. Note that $\perp \notin D$, so a functor $f : (D \rightarrow CP(D_\perp))$ is not defined on $\perp$, while multisets $A \in CP(D_\perp)$ may contain $\perp$. For this reason it is necessary that lifting specify explicitly the action of $f^\dagger$ and $\eta^\dagger$ on $\perp$.

$$f^\dagger(A) \overset{\text{def}}{=} \biguplus_{a \in A} f(a) \uplus \biguplus_{\perp \in A} \{\perp\}$$

$$f^\dagger(G : A \longrightarrow B) \overset{\text{def}}{=} \biguplus_{\substack{\langle a, b\rangle \in G \\ a \neq \perp}} f(a \longrightarrow b) \uplus \biguplus_{\substack{\langle \perp, b\rangle \in G \\ b \neq \perp}} \biguplus_{y \in f(b)} \langle \perp, y\rangle \uplus \biguplus_{\langle \perp, \perp\rangle \in G} \{\langle \perp, \perp\rangle\}$$

The above describes what lifting does to objects (which in this case are functors). We now describe what lifting does to arrows (which are natural transformations). Recall that natural transformations are maps from objects to arrows.

Given $\eta : f \longrightarrow g$, we have $\eta^\dagger : f^\dagger \longrightarrow g^\dagger$, given by

$$\eta^\dagger(A) \stackrel{\text{def}}{=} \biguplus_{\substack{a\in A\\ a\neq\perp}} \eta(a) \uplus \biguplus_{\perp\in A} \{\langle\perp,\perp\rangle\}.$$

Two important properties of lifting are that $g^\dagger$ is ω-colimit preserving if g is, and that $(\cdot)^\dagger$ is itself an ω-colimit preserving functor.

Sequential composition $\cdot\,;\cdot$: $f;g$ is shorthand for $g^\dagger \circ f$, and as such it inherits the desirable ω-colimit preservation properties from lifting.

We now describe a semantic function $\mathcal{D}[\![\cdot]\!] : \mathbf{Com} \to (D \to CP(D_\perp))$. Note that since D is completely "flat", when it is interpreted as a category the only arrows are the identities. Thus, when we describe the functors in the semantics we will specify only their action on objects, since functors by definition preserve identity arrows. If we were to give a semantics for a non-flat domain, we would be required to explicitly specify the action of the semantic functors on the arrows of the domain, since in general this is not trivial. We use the variable d to refer to elements of D.

$\mathcal{D}[\![x :=?]\!] = \lambda d.D$

$\mathcal{D}[\![x := c]\!] = \lambda d.\{c\}$

$\mathcal{D}[\![x := x-1]\!] = \lambda d.\{d-1\}$

$\mathcal{D}[\![x := x+1]\!] = \lambda d.\{d+1\}$

$\mathcal{D}[\![\mathsf{skip}]\!] = \lambda d.\{d\}$

$\mathcal{D}[\![S_1;S_2]\!] = \mathcal{D}[\![S_1]\!];\mathcal{D}[\![S_2]\!]$

$$\mathcal{D}[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!] = \lambda d.\begin{cases} \mathcal{D}[\![S_1]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{true} \\ \mathcal{D}[\![S_2]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{false} \end{cases}$$

We want to define the meaning of a while loop as a colimit, and since the meaning is a functor, it must be the colimit of a diagram in the functor category $D \rightarrow CP(D_\perp)$. This is the category whose objects are functors $D \rightarrow CP(D_\perp)$ and whose arrows are natural transformations. The initial object is the functor $\lambda d.\{\perp\}$, as is easily verified. We will call this Ω. Define

$$\mathcal{D}[\![\textsf{while } B \textsf{ do } S \textsf{ od}]\!] = \text{colim}(W_0 \xrightarrow{\eta_0} W_1 \xrightarrow{\eta_1} W_2 \xrightarrow{\eta_2} \cdots)$$

where W_i and η_i are defined as follows:

$W_0 = \Omega = \lambda d.\{\perp\}$

For $n \geq 1$: $W_n = \lambda d. \begin{cases} (\mathcal{D}[\![S]\!]; W_{n-1})d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{true} \\ \mathcal{D}[\![\textsf{skip}]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{false} \end{cases}$

$$\eta_0 x : W_0 x \longrightarrow W_1 x = \begin{cases} \biguplus_{x' \in \mathcal{D}[\![S]\!]x} \{\langle \perp, \perp \rangle\} & \text{if } \mathcal{B}[\![B]\!]x = \textsf{true} \\ \{\langle \perp, x \rangle\} & \text{if } \mathcal{B}[\![B]\!]x = \textsf{false} \end{cases}$$

For $n \geq 1$: $\eta_n x : W_n x \longrightarrow W_{n+1} x =$

$$\begin{cases} \biguplus_{\substack{x' \in \mathcal{D}[\![S]\!]x \\ x' \neq \perp}} \eta_{n-1} x' \uplus \biguplus_{\perp \in \mathcal{D}[\![S]\!]x} \{\langle \perp, \perp \rangle\} & \text{if } \mathcal{B}[\![B]\!]x = \textsf{true} \\ \{\langle x, x \rangle\} & \text{if } \mathcal{B}[\![B]\!]x = \textsf{false} \end{cases}$$

The colimit is determined pointwise; i.e. for all d, $\mathcal{D}[\![\textsf{while } B \textsf{ do } S \textsf{ od}]\!]d =$ $\text{colim}(W_0 d \xrightarrow{\eta_0 d} W_1 d \xrightarrow{\eta_1 d} W_2 d \xrightarrow{\eta_2 d} \cdots)$.

2.5 Some Examples

Example 2.1.

$\mathcal{D}[\![\textsf{while true do skip od}]\!]x =$

$$\text{colim}((\lambda d.\{\perp\})x \rightarrow (\lambda d.\{d\}; \lambda d.\{\perp\})x \rightarrow (\lambda d.\{d\}; \lambda d.\{d\}; \lambda d.\{\perp\})x \rightarrow \cdots)$$

$$= \text{colim}(\{\perp\} \longrightarrow \{\perp\} \longrightarrow \{\perp\} \longrightarrow \cdots)$$

$$= \{\perp\}$$

∎

Thus, $\mathcal{D}[\![\textsf{while true do skip od}]\!] = \lambda d.\{\perp\} = \Omega$. This is a pleasant property, since it means we can replace Ω in the definition of $\mathcal{D}[\![\textsf{while } B \textsf{ do } S \textsf{ od}]\!]$ by $\mathcal{D}[\![\textsf{loop}]\!]$, where loop $\equiv$ while true do skip od.

Example 2.2. $\mathcal{D}[\![x := 0]\!]d = \{0\}$ ∎

Example 2.3.

$$\begin{aligned}
&\mathcal{D}[\![\textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!]d = \\
&\quad \mathrm{colim}(\overbrace{\{\perp\} \longrightarrow \{\perp\} \longrightarrow \cdots \{\perp\}}^{d \text{ times}} \longrightarrow \{0\} \longrightarrow \{0\} \longrightarrow \cdots) \\
&= \{0\}
\end{aligned}$$

∎

Thus, $\mathcal{D}[\![\textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!] = \lambda d.\{0\} = \mathcal{D}[\![x := 0]\!]$.

Example 2.4. $\mathcal{D}[\![x :=?; \textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!] =$

$$\begin{array}{llllll}
\mathrm{colim}(& \{\perp, & \perp, & \perp, & \ldots & \} \\
 & \{0, & \perp, & \perp, & \ldots & \} \\
 & \{0, & 0, & \perp, & \ldots & \} \\
 & & & \cdots) & &
\end{array}$$

$= \{0, 0, 0, \ldots\}$ ∎

Of course, we should have expected this, since

$$\begin{aligned}\mathcal{D}[\![x :=?; \textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!] &= \mathcal{D}[\![x :=?; x := 0]\!] \\ &= (\lambda d.\{0\})^{\dagger} \circ (\lambda d.D) \\ &= \biguplus_{d \in D} \{0\}.\end{aligned}$$

This may seem unnatural, but it is a necessary effect of our approach that $\mathcal{D}[\![x :=?; \textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!]$ is different from $\mathcal{D}[\![x := 0]\!]$. The use of multisets and functors provides a more detailed description of the approximations, but at the same time can increase the cardinality of the approximate objects to account for indeterminate behavior. The semantics of programs over the determinate portion of our language (i.e. without "$x :=?$") will always be a singleton, and will satisfy all the usual semantic relations.

The previous example also serves to point out how our semantics differs from those considered by Apt and Plotkin. In their proof they consider the same statement as above, but in a domain theoretic setting. Without arrows for approximation or multiset objects, they show

$$\begin{aligned}\mathcal{D}[\![x :=?; \textsf{while } x > 0 \textsf{ do } x := x - 1 \textsf{ od}]\!] &= \bigsqcup \Big\{\{0, \perp\}, \{0, \perp\}, \{0, \perp\}, \ldots\Big\} \\ &= \{0, \perp\}.\end{aligned}$$

This clearly does not agree with the operational behavior, therefore full abstraction must fail. In our semantics, $\perp$ is not a possibility in the limit, and hence except for the multiplicity of the result, we can achieve full abstraction. This is the subject of the next section.

2.6 Relationship with the Operational Semantics

In this section we begin by investigating properties of the operations semantics $Op[\![\,]\!]$, ultimately proving its compositionality. We then demonstrate that an abstraction of the denotational semantics $\mathcal{D}[\![\,]\!]$ exactly coincides with $Op[\![\,]\!]$. From these, the full-abstraction of the abstracted semantics follows as a corollary.

Lemma 2.3. For all statements S and states d,

$$Op[\![S]\!]d = \bigcup_{\substack{S',d' \text{ s.t.} \\ \langle S, d\rangle \to \langle S', d'\rangle}} Op[\![S']\!]d' \cup \{d' \mid \langle S, d\rangle \to d'\}.$$

Proof: Straightforward from the transition relations. ∎

Lemma 2.4. For all statements S_1, S_2, and states d,

$$Op[\![S_1; S_2]\!]d = \bigcup_{\substack{d' \in Op[\![S_1]\!]d \\ d' \neq \bot}} Op[\![S_2]\!]d' \cup \{\bot \mid \bot \in Op[\![S_1]\!]d\}.$$

Proof: Straightforward from the transition relations. ∎

Theorem 2.5. If $Op[\![S]\!] = Op[\![S']\!]$, then for all contexts $C[\cdot]$, we have

$$Op[\![C[S]]\!] = Op[\![C[S']]\!].$$

Proof: The proof is a structural induction on the context $C[\cdot]$.

Case $C \equiv \textsf{if } B \textsf{ then } [\cdot] \textsf{ else } T \textsf{ fi}$: From Lemma 2.3 it follows that, for all d,

$$\begin{aligned}
Op[\![\textsf{if } B \textsf{ then } S \textsf{ else } T \textsf{ fi}]\!]d &= \begin{cases} Op[\![S]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{true} \\ Op[\![T]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{false} \end{cases} \\
\text{(by hypothesis)} &= \begin{cases} Op[\![S']\!]d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{true} \\ Op[\![T]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \textsf{false} \end{cases} \\
\text{(by Lemma)} &= Op[\![\textsf{if } B \textsf{ then } S' \textsf{ else } T \textsf{ fi}]\!]d
\end{aligned}$$

Case $C \equiv \textsf{if } B \textsf{ then } T \textsf{ else } [\cdot] \textsf{ fi}$: Similar to above.

Case $C \equiv [\cdot]; T$: We know from Lemma 2.4 that

$$\begin{aligned}
Op[\![S;T]\!]d &= \bigcup_{\substack{d' \in Op[\![S]\!]d \\ d' \neq \perp}} Op[\![T]\!]d' \cup \{\perp \mid \perp \in Op[\![S]\!]d\} \\
\text{(by hyp.)} &= \bigcup_{\substack{d' \in Op[\![S']\!]d \\ d' \neq \perp}} Op[\![T]\!]d' \cup \{\perp \mid \perp \in Op[\![S']\!]d\} \\
\text{(by Lemma)} &= Op[\![S';T]\!]d
\end{aligned}$$

Case $C \equiv T; [\cdot]$: As above, we know that

$$\begin{aligned}
Op[\![T;S]\!]d &= \bigcup_{\substack{d' \in Op[\![T]\!]d \\ d' \neq \perp}} Op[\![S]\!]d' \cup \{\perp \mid \perp \in Op[\![T]\!]d\} \\
\text{(by hyp.)} &= \bigcup_{\substack{d' \in Op[\![T]\!]d \\ d' \neq \perp}} Op[\![S']\!]d' \cup \{\perp \mid \perp \in Op[\![T]\!]d\} \\
\text{(by Lemma)} &= Op[\![T;S']\!]d
\end{aligned}$$

Case $C \equiv \mathsf{while}\ B\ \mathsf{do}\ [\cdot]\ \mathsf{od}$: We show that for all d,

$$Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d \subseteq Op[\![\mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od}]\!]d.$$

By symmetry, the reverse must also be true, and the desired equality follows.

Case 1: If $\mathcal{B}[\![B]\!]d = \mathsf{false}$, then

$$Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d = Op[\![\mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od}]\!]d = \{d\}$$

Case 2: Suppose $\mathcal{B}[\![B]\!]d = \mathsf{true}$, and let $d' \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$, $d' \neq \bot$. Then $\exists n \geq 1$ and a sequence $d = d_0, d_1, \ldots, d_n = d'$ such that

$$d_{i+1} \in Op[\![S]\!]d_i \text{ for } i = 0, \ldots, n-1,$$

$$\mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for } i = 0, \ldots, n-1,$$

$$\text{and } \mathcal{B}[\![B]\!]d_n = \mathsf{false}.$$

By the hypothesis, this means

$$d_{i+1} \in Op[\![S']\!]d_i \text{ for } i = 0, \ldots, n-1,$$

$$\mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for } i = 0, \ldots, n-1,$$

$$\text{and } \mathcal{B}[\![B]\!]d_n = \mathsf{false},$$

which is equivalent to

$$d' \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od}]\!]d.$$

Case 3: Suppose $\mathcal{B}[\![B]\!]d = \mathsf{true}$, and $\bot \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$. Then there exists a sequence $d = d_0, d_1, \ldots$ such that

$$d_{i+1} \in Op[\![S]\!]d_i \text{ for all } i,$$

$$\text{and } \mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for all } i.$$

By the hypothesis, this means

$$d_{i+1} \in Op[\![S']\!]d_i \text{ for all } i,$$
$$\text{and } \mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for all } i,$$

or equivalently,

$$\perp \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S'\ \mathsf{od}]\!]d.$$

∎

Definition Let $P_S(D_\perp)$ be the Smyth powerdomain of $D_\perp$. Viewed as a category, the objects of $P_S(D_\perp)$ are sets of elements of $D_\perp$, and there is an arrow $A \longrightarrow B$ if and only if $A \sqsubseteq_S B$. $P_S(D_\perp)$ is the result of collapsing the objects of $CP(D_\perp)$ from multisets to sets, and of collapsing parallel arrows to a single one in $P_S(D_\perp)$. We define $ab : CP(D_\perp) \rightarrow P_S(D_\perp)$ as the obvious abstraction functor, taking multiset objects of $CP(D_\perp)$ to their corresponding set objects of $P_S(D_\perp)$, and taking arrows of $CP(D_\perp)$ to the corresponding arrows of $P_S(D_\perp)$. ab is easily seen to preserve composition and identity arrows.

Note that ab is not an ω-colimit preserving functor (as can be seen by applying ab to the sets in Example 2.4). This is the only aspect of our semantics that is not "continuous".

Theorem 2.6. $ab(\mathcal{D}[\![S]\!]) = Op[\![S]\!]$ for all commands S.

Proof: The proof is a structural induction by cases.

Case $S \equiv A$: For the atomic commands A, it is trivially the case that

$$Op[\![A]\!]d = ab(\mathcal{D}[\![A]\!]d) \text{ for all } d \text{ in } D.$$

Case $S \equiv \mathsf{skip}$: Also trivially,

$$Op[\![\mathsf{skip}]\!]d = \{d\} = ab(\mathcal{D}[\![\mathsf{skip}]\!]d) \text{ for all } d \text{ in } D.$$

Case $S \equiv \mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$: From Lemma 2.3 it follows that, for all d,

$$\begin{aligned}
Op[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]d &= \begin{cases} Op[\![S_1]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{true} \\ Op[\![S_2]\!]d & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{false} \end{cases} \\
\text{(by ind. hyp.)} &= \begin{cases} ab(\mathcal{D}[\![S_1]\!]d) & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{true} \\ ab(\mathcal{D}[\![S_2]\!]d) & \text{if } \mathcal{B}[\![B]\!]d = \mathsf{false} \end{cases} \\
&= ab(\mathcal{D}[\![\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}]\!]d)
\end{aligned}$$

Case $S \equiv S_1; S_2$: From the definition of sequential composition we have

$$\begin{aligned}
\mathcal{D}[\![S_1; S_2]\!]d &= (\mathcal{D}[\![S_2]\!])^\dagger(\mathcal{D}[\![S_1]\!]d) \\
&= \biguplus_{\substack{d' \in \mathcal{D}[\![S_1]\!]d \\ d' \neq \perp}} \mathcal{D}[\![S_2]\!]d' \uplus \biguplus_{\perp \in \mathcal{D}[\![S_1]\!]d} \{\perp\}.
\end{aligned}$$

This means that, for all d,

$$ab(\mathcal{D}[\![S_1; S_2]\!]d) = \bigcup_{\substack{d' \in ab(\mathcal{D}[\![S_1]\!]d) \\ d' \neq \perp}} ab(\mathcal{D}[\![S_2]\!]d') \cup \{\perp \mid \perp \in ab(\mathcal{D}[\![S_1]\!]d)\}$$

$$\begin{aligned}
\text{(by ind. hyp.)} &= \bigcup_{\substack{d' \in Op[\![S_1]\!]d \\ d' \neq \bot}} Op[\![S_2]\!]d' \cup \{\bot | \bot \in Op[\![S_1]\!]d\} \\
\text{(by Lemma 2.4)} &= Op[\![S_1; S_2]\!]d
\end{aligned}$$

<u>Case $S \equiv$ while B do S od:</u> This case proceeds via two lemmas.

Lemma 2.7. For all d, $Op[\![\text{while } B \text{ do } S \text{ od}]\!]d \subseteq ab(\mathcal{D}[\![\text{while } B \text{ do } S \text{ od}]\!]d)$

Proof: Let $d' \in Op[\![\text{while } B \text{ do } S \text{ od}]\!]d$.

Case 1: If $\mathcal{B}[\![B]\!]d = \text{false}$, then $d = d'$, and

$$\begin{aligned}
\mathcal{D}[\![\text{while } B \text{ do } S \text{ od}]\!] &= \text{colim}(W_0 d \to W_1 d \to \ldots) \\
&= \text{colim}(\{\bot\} \to \{d\} \to \ldots) \\
&= \{d\},
\end{aligned}$$

so $d' \in ab(\mathcal{D}[\![\text{while } B \text{ do } S \text{ od}]\!]d)$.

Case 2: Suppose $\mathcal{B}[\![B]\!]d = \text{true}$, and $d' \neq \bot$. Then $\exists n \geq 1$ and a sequence $d = d_0, d_1, \ldots, d_n = d'$ such that

$d_{i+1} \in Op[\![S]\!]d_i$ (equivalently $\langle S, d_i \rangle \to^* d_{i+1}$) for $i = 0, \ldots, n-1$,

$\mathcal{B}[\![B]\!]d_i = \text{true}$ for $i = 0, \ldots, n-1$,

and $\mathcal{B}[\![B]\!]d_n = \text{false}$.

Now note that if $\mathcal{B}[\![B]\!]d_i = \text{true}$, then for any $j \geq 1$

$$\begin{aligned}
W_j d_i &= (\mathcal{D}[\![S]\!]; W_{j-1})d_i \\
&= \biguplus_{\substack{e \in \mathcal{D}[\![S]\!]d_i \\ e \neq \bot}} W_{j-1}e \uplus \{\bot | \bot \in \mathcal{D}[\![S]\!]d_i\},
\end{aligned}$$

so given that $d_{i+1} \in Op[\![S]\!]d_i$ implies that $d_{i+1} \in \mathcal{D}[\![S]\!]d_i$ by the induction hypothesis, we have that

$$W_{j-1}d_{i+1} \subseteq W_j d_i, \text{ for } i < n.$$

Also, $\mathcal{B}[\![B]\!]d_n = \mathsf{false}$ implies that

$$W_j d_n = \{d_n\} \text{ for any } j > 0.$$

This, together with the chain $d = d_0, d_1, \ldots, d_n = d'$ defined above, gives us

$$\{d'\} = \{d_n\} = W_1 d_n \subseteq W_2 d_{n-1} \subseteq \cdots \subseteq W_n d_1 \subseteq W_{n+1} d_0 = W_{n+1} d.$$

Now, if we look at the definition of the natural transformation η_j, for $j > 0$, we see that

$$\begin{aligned} d' \neq \bot,\ d' \in W_{n+1} \quad &\Rightarrow \quad d' \in W^* d \\ &\Leftrightarrow \quad d' \in \mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d \\ &\Rightarrow \quad d' \in ab(\mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d). \end{aligned}$$

Case 3: Suppose $\mathcal{B}[\![B]\!]d = \mathsf{true}$, and $d' = \bot$. Then there exists a sequence $d = d_0, d_1, \ldots$ such that

$$\begin{gathered} d_{i+1} \in Op[\![S]\!]d_i \text{ for all } i, \\ \text{and } \mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for all } i. \end{gathered}$$

We wish to show that $\mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$ contains $\bot$, or (by unfolding the definition) that the diagram

$$W_0 d_0 \xrightarrow{\eta_0 d_0} W_1 d_0 \xrightarrow{\eta_1 d_0} W_2 d_0 \xrightarrow{\eta_2 d_0} \cdots$$

contains an infinite chain $\{\perp\} \longrightarrow \{\perp\} \longrightarrow \cdots$.

Noting that by the induction hypothesis, $d_{i+1} \in Op[\![S]\!]d_i$ for all i, it is a straightforward induction to show that, for any $n \geq 0$

$$\eta_n d_j : W_n d_j \longrightarrow W_{n+1} d_j \ni \langle \perp, \perp \rangle \text{ for all } j \geq 0.$$

$\mathcal{B}[\![B]\!]d_j = \mathsf{true}$ for all $j \geq 0$ implies

$$\eta_0 d_j : W_0 d_j \longrightarrow W_1 d_j \ni \langle \perp, \perp \rangle \text{ for all } j \geq 0$$

from the definition of η_0. For $n > 1$, we have that, for all $j \geq 0$

$$\begin{aligned} \eta_n d_j : W_n d_j \longrightarrow W_{n+1} d_j &\ni \biguplus_{x \in \mathcal{D}[\![S]\!]d_j} \eta_{n-1} x \\ (\text{since } d_{j+1} \in \mathcal{D}[\![S]\!]d_j) &\ni \eta_{n-1} d_{j+1} \\ (\text{by induction}) &\ni \langle \perp, \perp \rangle. \end{aligned}$$

Thus,

$$\begin{aligned} d' = \perp &\in \operatorname{colim}(W_0 d_0 \xrightarrow{\eta_0 d_0} W_1 d_0 \xrightarrow{\eta_1 d_0} W_2 d_0 \xrightarrow{\eta_2 d_0} \cdots) \\ &= \mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d, \end{aligned}$$

and we have $d' \in ab(\mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d)$. ∎

Lemma 2.8. For all d, $ab(\mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d) \subseteq Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$

Proof: Let $d' \in \mathcal{D}[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$. Then

$$d' \in \operatorname{colim}(W_0 d \rightarrow W_1 d \rightarrow \cdots) \Rightarrow d' = \sqcup Q,$$

where Q is a chain through the diagram, as defined in Theorem 2.2. From the definition of the W_i and the η_i, we know that the only distinct elements of Q are $\perp$ and d'. Thus, $d' \in Q$, which implies $d' \in W_n d$ for some finite n. Let m be the least such index. Now, assuming $d' \neq \perp$, there is a sequence $d = d_0, d_1, \ldots, d_m = d'$ such that

$$d_{i+1} \in \mathcal{D}[\![S]\!]d_i \text{ for } i = 0, \ldots, m-1,$$
$$\mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for } i = 0, \ldots, m-1,$$
$$\text{and } \mathcal{B}[\![B]\!]d_m = \mathsf{false}.$$

But this means that (by induction hypothesis)

$$d_{i+1} \in Op[\![S]\!]d_i \text{ for } i = 0, \ldots, m-1,$$
$$\mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for } i = 0, \ldots, m-1,$$
$$\text{and } \mathcal{B}[\![B]\!]d_m = \mathsf{false}.$$

This we know is equivalent to $d' = d_m \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$.

If $d' = \perp$, then there is an infinite sequence $d = d_0, d_1, \ldots$ such that

$$d_{i+1} \in \mathcal{D}[\![S]\!]d_i \text{ for all } i,$$
$$\text{and } \mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for all } i.$$

But this means that (by induction hypothesis)

$$d_{i+1} \in Op[\![S]\!]d_i \text{ for all } i,$$
$$\text{and } \mathcal{B}[\![B]\!]d_i = \mathsf{true} \text{ for all } i,$$

which implies $\perp = d' \in Op[\![\mathsf{while}\ B\ \mathsf{do}\ S\ \mathsf{od}]\!]d$. ∎

Corollary 2.9 (Full-abstraction) For all statements S and S' we have

$$ab(\mathcal{D}[\![S]\!]) = ab(\mathcal{D}[\![S']\!])$$

if and only if

$$Op[\![C[S]]\!] = Op[\![C[S']]\!] \text{ for all contexts } C[\cdot].$$

Proof: $ab(\mathcal{D}[\![S]\!]) = ab(\mathcal{D}[\![S']\!])$ is equivalent to $Op[\![S]\!] = Op[\![S']\!]$ by Theorem 2.6. If we take $C[\cdot]$ to be the empty context, we see that $Op[\![C[S]]\!] = Op[\![C[S']]\!]$ implies that $Op[\![S]\!] = Op[\![S']\!]$; Theorem 2.5 gives us the reverse implication. ∎

These results are the main results of this chapter. The primary achievement is that this semantics generalizes the usual notion of continuous least-fixed-point semantics. It is continuous in the category-theoretic sense, and it is fully abstract via an abstraction functor from multisets to sets. The failure of continuity required by Apt and Plotkin's result is isolated to this abstraction functor. The framework we have developed here is general enough to accommodate domains that are not "flat", and even Lehmann's categorically generalized domains.

In the next chapter we begin to look at a more general setting than sequential languages, one better suited to studying the problems of unbounded indeterminacy.

Chapter 3

Dataflow Networks and Indeterminacy

Beginning with this chapter, we change the setting of our study. In this chapter we introduce and describe a different programming paradigm called dataflow networks. Dataflow networks provide a general framework for studying the interaction of concurrency and indeterminacy, and provide a more interesting setting than sequential languages for discussing the problems of semantic models for indeterminacy.

In the first section we present the necessary background, definitions, and operational semantics of dataflow networks. In the second section we describe Kahn's principle, which is an elegant semantic model for determinate dataflow networks. Finally, in the third section we introduce indeterminacy into the dataflow context.

For the remainder of this thesis we will work in the dataflow setting, and study the problem of extending the work of Kahn by developing good semantic models for indeterminate dataflow networks.

3.1 Dataflow Networks

3.1.1 Background and Definitions

In this section we briefly review the definitions and terminology of dataflow networks. Since the main point of this thesis is to investigate relations between semantic models and observable properties of networks, in this section we only give an informal presentation of the necessary background. See, for example, [Sta89a,PS88,Sta87,JK88,Jon89,Sha90] for the formal development on which this is based.

The fundamental unit of a network is a type of state transition machine called an *port automaton.* This type of automaton is actually a special case of the *input-output automata* described by Lynch and Tuttle [LT87]. Port automata communicate with each other and the outside world by sending and receiving data values (or "tokens") on "ports". Each port is either an input or an output port for the automaton, and in each step of its execution an automaton may poll or read an input port, write to an output port, or change its internal state. The ports of the automata are buffered, meaning that the tokens may be arbitrarily delayed entering (or leaving) the input (or output) ports, and hence an external observer cannot distinguish the relative order of unrelated events on separate channels. Furthermore, the automata are required to be receptive, meaning that no automaton can at any time prevent itself from receiving input tokens. It is important to note that an input-output port automaton may have an infinite number of states, and the domain of data values may be infinite, so there is no restriction on the function the automaton may compute on the data values.

A *dataflow network* consists of a set of concurrently executing port automata connected together by directed channels. The channels act as perfect, unbounded FIFO queues. Each channel may be connected to at most one input port, and at most one output port. There are three types of channels: *input channels*, which are not connected to an output port of any automaton, and transmit data into the network from outside; *output channels*, which are not connected to an input port of any automaton, and transmit data from the network outside; and *internal channels*, which transmit data between network nodes.

An important feature of dataflow networks is that they can be composed, and larger networks can be built using smaller networks in place of individual automata. As with the port automata, networks to be composed must satisfy the compatibility condition that each channel may occur at most once as an input channel and at most once as an output channel. The definitions of input and output channels remain the same (though in the process of composition an external channel of a component network may become an internal channel of the larger network). For a network N, we denote its set of input channels by I_N and its set of output channels by O_N.

There are two atomic operations of network composition: aggregation and looping. The aggregate of networks M and N, written $M||N$, is the network formed by combining them 'side-by-side' with no identification of channels. Given a network M, $\mathsf{loop}(a, b, M)$ is the network formed from M by identifying input channel a with output channel b. It is clear that any network can be constructed from these operations.

Example 3.1. Figure 3.1 illustrates the aggregation and looping operations.

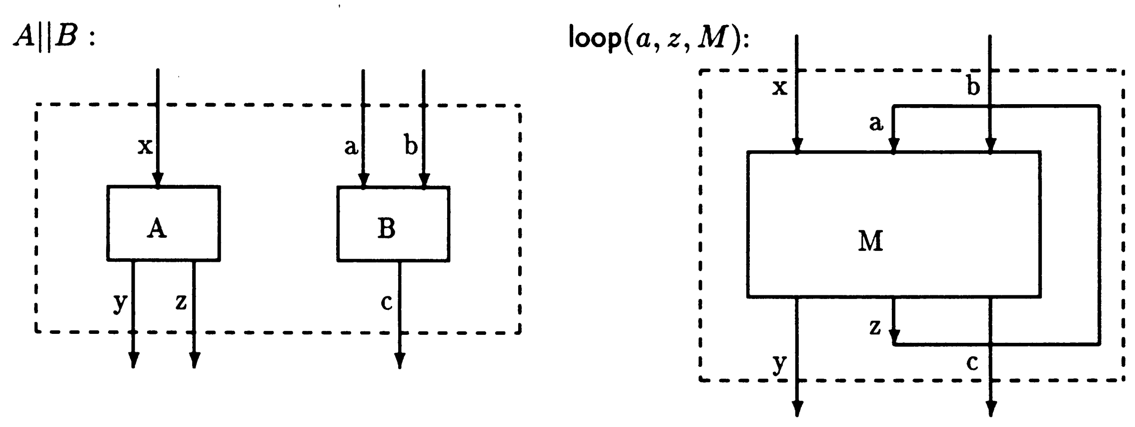

Figure 3.1: The aggregation and looping operations of network composition

The diagram on the left shows the aggregate of a network A with $I_A = \{x\}$, $O_A = \{y, z\}$, together with a network B with $I_B = \{a, b\}$, $O_B = \{c\}$. The aggregate network $A||B$ has $I_{A||B} = \{x, a, b\}$, $O_{A||B} = \{y, z, c\}$. The diagram on the right shows the network $\mathsf{loop}(a, z, M)$, where $I_M = \{x, a, b\}$ and $O_M = \{y, z, c\}$. The looped network has $I_{\mathsf{loop}(a,z,M)} = \{x, b\}$ and $O_{\mathsf{loop}(a,z,M)} = \{y, c\}$. Note that if M is $A||B$, then the result is a network consisting of subnetworks A and B, with output channel z of A connected to input channel a of B. ∎

For the remainder of this thesis, we will only explicitly label channels when necessary for clarity. Also, we will refer to particularly simple networks or automata simply as processes, and we will typically describe them as if they were programmed in a simple sequential language (rather than by describing an automaton).

A *computation* of a network is a linear sequence of state transitions of the component automata that captures a "complete" run of the network. The formal condition corresponding to the intuitive notion of a complete run is nontrivial to state; see Panangaden and Stark [PS88] for a description. We define *communication events* as transitions of a computation in which data either arrives on an input channel or is sent along an output channel (*input events* and *output events*, respectively). A *trace* of a network is the sequence of communications events of a computation of the network. Traces are commonly written as sequences of pairs $\langle$*channel_name*, *data_value*$\rangle$. For a network N, we define $\Gamma[\![N]\!]$ as the set of computations of N, and $\mathcal{T}[\![N]\!]$ as the set of traces of N. If Γ is a computation of N, and C is a set of channels, we define $\Gamma\lceil_C$ as the subsequence of Γ consisting only of events on channels in C. Similarly, for a trace T of N, we define $T\lceil_C$ as the subsequence of T consisting only of events on channels in C.

We call the sequence of values of the input or output events on a channel the *history* of the channel. We write $H_c(\Gamma)$ for the sequence of values passed over the channel c in the computation Γ – the history of channel c in the computation. Similarly, for a trace T we define $H_c(T)$ as the sequence of values from all the events on channel c in T. For a set of channels C, we define $H_C(\Gamma)$ (similarly $H_C(T)$) as the C-indexed tuple of histories of the channels in C.

Example 3.2. Consider the network shown in Figure 3.2. The process *LR-merge*

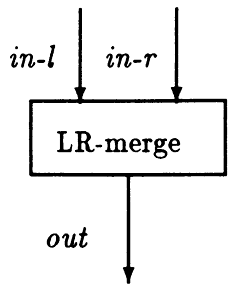

Figure 3.2: An example network

has $I_{LR\text{-}merge} = \{in\text{-}l, in\text{-}r\}$ and $O_{LR\text{-}merge} = \{out\}$. It merges its two inputs by first reading channel *in-l* and outputting the token it gets (if there is one) on channel *out*, then reading channel *in-r* and outputting the token it gets on channel *out*, and repeating indefinitely. If *LR-merge* tries to read a channel that is empty, it waits (possibly forever) for data to arrive.

The following are two possible traces of this network:

$$\langle in\text{-}r, 1\rangle\langle in\text{-}l, 2\rangle\langle in\text{-}r, 3\rangle\langle out, 2\rangle\langle out, 1\rangle$$

$$\langle in\text{-}l, 2\rangle\langle out, 2\rangle\langle in\text{-}r, 1\rangle\langle out, 1\rangle\langle in\text{-}r, 3\rangle$$

Both of these traces have channel histories

$$H_{I_{LR\text{-}merge}}(T) = \langle 2, 1\,3\rangle, \quad \text{and}$$

$$H_{O_{LR\text{-}merge}}(T) = 2\,1.$$

∎

3.1.2 Operational Semantics

The *input-output relation* of a network N is the set of all pairs of input and output channel histories, $(H_{I_N}(\Gamma), H_{O_N}(\Gamma))$, with $\Gamma \in \Gamma[\![N]\!]$. The input-output relation of a network is what we consider to be the "observable" behavior. From the outside, an observer can see the values of the input and output channels, but cannot distinguish the relative order of the input events, output events, and internal events. Note that we consider the full, possibly infinite, streams of values to be observable. Other, more restrictive notions are possible, as in the work of Rabinovich and Trakhtenbrot [RT88], who consider a theory based on finite observations (we discuss their work further in Chapter 7). We write $\mathcal{IO}[\![N]\!]$ for the input-output relation of the network N, and since this is the observable behavior we have $Op[\![N]\!] = \mathcal{IO}[\![N]\!]$ for all dataflow networks N.

We now remind the reader of some of the properties useful in relating abstract semantics of dataflow networks to observable operational behavior. We say two networks N_1 and N_2 are *operationally equal* if, for every network context $\mathcal{C}[\,]$, the composite networks $\mathcal{C}[N_1]$ and $\mathcal{C}[N_2]$ have the same input-output relation, i.e. that $\mathcal{IO}[\![\mathcal{C}[N_1]]\!] = \mathcal{IO}[\![\mathcal{C}[N_2]]\!]$.

A semantic model $\mathcal{D}[\![\,]\!]$ is *adequate* if whenever $\mathcal{D}[\![N_1]\!] = \mathcal{D}[\![N_2]\!]$, N_1 and N_2 are operationally equal. A semantic model $\mathcal{D}[\![\,]\!]$ is *fully abstract* if the converse of adequacy holds as well, i.e. that $\mathcal{D}[\![N_1]\!] = \mathcal{D}[\![N_2]\!]$ if and only if N_1 and N_2 are operationally equal.

3.2 Kahn's Principle

The first major work in the area of dataflow semantics was by Kahn [Kah77], who gave a simple and elegant semantics for dataflow networks in which all the processes are determinate. Processes are determinate when there is only one possible stream of outputs for every stream of data tokens input to the network, and his semantics describes networks as continuous stream-valued functions corresponding to the (functional) input-output relation of the network. Specifically, a network M with m input channels and n output channels is represented as a function $f_M : \mathsf{S}^m \rightarrow \mathsf{S}^n$, where S represents the domain of streams over the underlying domain of data tokens. If N is a network with m' inputs and n' outputs, then $f_{M||N} = \langle f_M, f_N \rangle : \mathsf{S}^{m+m'} \rightarrow \mathsf{S}^{n+n'}$, where $\langle f_M, f_N \rangle$ is the function that on input $\langle i, i' \rangle \in \mathsf{S}^{m+m'}$ (with $i \in \mathsf{S}^m, i' \in \mathsf{S}^{m'}$) produces output $\langle f_M(i), f_N(i') \rangle \in \mathsf{S}^{n+n'}$. Similarly, $f_{\mathsf{loop}(a,b,M)} = \mathsf{fix}(a, b, f_M)$, where $g = \mathsf{fix}(a, b, f_M)$ is the function in $\mathsf{S}^{m-1} \rightarrow \mathsf{S}^{n-1}$ that computes fixed points of f_M; $g(i) = o$ means that o is the least output on the components other than b such that there exists a stream l with $f_M(i, l) = (o, l)$ (where l is actually the ath component of the input and the bth component of the output). A nice presentation and proof of Kahn's principle is given by Lynch and Stark [LS89]

Kahn's semantics has two particularly desirable properties that we focus on for this thesis. The first is that it is fully abstract, which is a direct consequence of the fact that the denotational representation by functions coincides with the operational semantics. The second is a fixed-point property; that the denotation of a composite network can be obtained from the denotations of its components as the least fixed point of the recursive equations describing the network, providing us with a simple method for computing the meaning of a looped network as a limit.

Example 3.3. Consider the network in Figure 3.3. *Switch* is a process that first

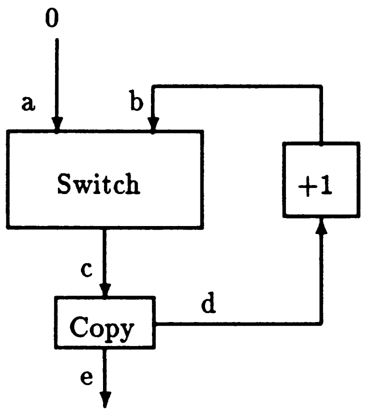

Figure 3.3: An example network

reads and passes to output channel c the first token from channel a (in this case a 0), and thereafter reads and passes everything from channel b. *Copy* copies every token from channel c to channels e and d. +1 is a process that continuously reads the value from channel d, and outputs that value plus one to channel b. These definitions generate the set of equations:

$$\begin{aligned} c &= cons(0, b) \\ e, d &= c \\ b &= +1(d) \end{aligned}$$

where a, b, c, d, e are variables representing streams, and $+1()$ is a stream function that produces a stream of values pointwise one greater than those of its argument. This system reduces to the recursive equation

$$e = cons(0, +1(e)),$$

which has least fixed-point solution $e = 0\,1\,2\,\cdots$. It is obvious from the descriptions that this equals the actual output of the network. ■

For the remainder of this thesis we will use Kahn's principle as a base, and study the problem of generalizing it to the setting of indeterminate dataflow networks. As we shall see, in the next section and beyond, the indeterminate dataflow setting is rich with complications – not the least of which is that "indeterminate dataflow" is by itself not well defined.

3.3 Indeterminate Dataflow Networks

Indeterminate dataflow networks are those for which the input-output relation is not functional. The most common examples of indeterminate networks are those containing the various *merge* primitives, e.g. fair merge, angelic merge, infinity-fair merge, and unfair merge. While for some time identifying such networks simply as indeterminate was assumed to have an unambiguous meaning, the work of Panangaden, Stark, and others [MPS88,Sta88,PS88,PS87] has shown that in fact there are many provably inequivalent indeterminate primitives.

We can relate indeterminate primitives by the class of operationally distinct networks constructible using them, and these authors have shown that when related in this way they form a hierarchy of several different levels. In particular, each of the merge primitives mentioned above (and described below) is in a distinct level of this indeterminate hierarchy.

3.3.1 The Indeterminate Merge Primitives

We now describe the various indeterminate merge primitives. We present informal descriptions of the processes, but these are sufficiently precise that the set of possible traces of each merge primitive is determined.

Fair merge was introduced by programmers as an abstraction of a fair scheduler. Like all the merge primitives, it is a process with two input channels and one output channel that produces on its output some interleaving of the tokens appearing on its inputs. Fair merge has the property that it guarantees that its output is a *shuffle* of its inputs, i.e. that the relative order of tokens on each input is preserved in the output, and that every token that is in either input will (eventually) be in the output. We define its input-output relation by $(\langle x, y\rangle, z) \in \mathcal{IO}[\![\mathit{Fair\ Merge}]\!]$ if and only if z is a shuffle of x and y. Fair merge is equivalent to having the ability to 'poll' a channel for the availability of data (like `select` in UNIX). We can think of fair merge having two separate "fairness" characteristics; it avoids trying to read data from an empty channel (which would result in a deadlock from waiting infinitely), and it avoids starvation of its inputs by assuring that both its inputs will be serviced eventually as long as they are not empty.

Angelic merge is a primitive that has only the first "fairness" characteristic of fair merge – it guarantees it will never try to read from an empty channel, but does not guarantee a fair servicing of both its channels. As a result, angelic merge has the property that if one of its inputs is finite, it will read all of the other input. Formally, its input-output relation is defined by $(\langle x, y\rangle, z) \in \mathcal{IO}[\![\textit{Angelic Merge}]\!]$ if and only if z is a shuffle of x' and y', where x' and y' are prefixes of x and y ($x' \sqsubseteq x$ and $y' \sqsubseteq y$) and

if x is finite, then $y' = y$;

if y is finite, then $x' = x$;

if both x and y are infinite, then $x' = x$ or $y' = y$ (or both).

Angelic merge is what we would get if we were to program a merge using McCarthy's ambiguity operator.

Angelic merge can be implemented by (is operationally equivalent to) a network built using fair merge and determinate processes, but the converse is provably not true, as shown in [PS88]. Thus, angelic merge is strictly less expressive than fair merge, meaning that the class of angelic merge networks is strictly contained in the class of fair merge networks. One of the ways of showing this difference is by noting that the input-output relation of any network built using angelic merge and determinate processes is monotone in the Hoare ordering, and the input-output relation of fair merge is not.

Infinity-fair merge is a primitive that has only the second "fairness" characteristic of fair merge – it guarantees it will service both input channels infinitely often, but does not guarantee that it will avoid the deadlock of trying to read from an empty channel. As a result, infinity-fair merge has the property that if one of its inputs is infinite, it will read all of the other input. Formally, its input-output relation is defined by $(\langle x, y\rangle, z) \in \mathcal{IO}[\![\mathit{Infinity\text{-}Fair\ Merge}]\!]$ if and only if z is a shuffle of prefixes $x' \sqsubseteq x$ and $y' \sqsubseteq y$ where

if x is infinite, then $y' = y$;

if y is infinite, then $x' = x$;

if both x and y are finite, then $x' = x$ or $y' = y$ (or both).

Infinity-fair merge is the type of merge we could program in a language with the ability to do repeated random integer assignment (like $x :=?$ from the previous chapter).

Infinity-fair merge can be implemented by a network built using angelic merge and determinate processes, but the converse is not true, as shown in [PS87]. Hence, infinity-fair merge is strictly less expressive than angelic merge and sits at a lower level of the hierarchy. As with the angelic merge and fair merge networks, the infinity-fair merge networks are separated from the angelic merge networks by a monotonicity property. The input-output relation of any network built from infinity-fair merge and determinate processes is monotone in the Egli-Milner ordering, but the input-output relation of angelic merge is not.

Unfair merge is the final indeterminate merge primitive we describe. It has neither of the "fairness" characteristics of fair merge, and can only guarantee that it will attempt to service some channel infinitely often, unless it deadlocks reading from an empty channel. Thus, the only property that unfair merge can guarantee is that it will read all of at least one of its inputs. Formally, its input-output relation is defined by $(\langle x, y\rangle, z) \in \mathcal{IO}[\![\textit{Unfair Merge}]\!]$ if and only if z is either a shuffle of x with a prefix $y' \sqsubseteq y$, or a shuffle of y with a prefix $x' \sqsubseteq x$. Unfair merge is what we could program in a language having the ability to do repeated bounded choice, as with an indeterminate "or".

Unfair merge can be implemented by a network built using infinity-fair merge, but the converse is not true. This is clear by noting that infinity-fair merge (and the primitives above it) embodies unbounded indeterminacy, while unfair merge is only boundedly indeterminate. Of course, unfair merge (and all the merge primitives) are strictly more expressive than only determinate processes.

The relationship among the classes of indeterminate dataflow networks defined by the different indeterminate merge primitives is illustrated in Figure 3.4. This diagram shows a pictorial representation of the hierarchy of indeterminate primitives, where the levels higher in the diagram contain those below them. Fair merge defines the highest known level of this indeterminate hierarchy and unfair merge defines the lowest level, although in [MPS88] the authors describe several unboundedly indeterminate primitives that lie between unfair merge and infinity-fair merge.

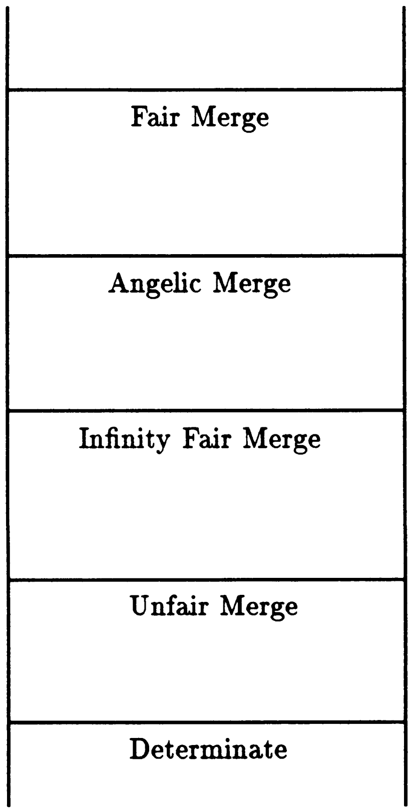

Figure 3.4: The hierarchy of indeterminate primitives

Chapter 4

Full Abstraction for Indeterminate Dataflow Networks

In this chapter we discuss the extension of Kahn's principle to dataflow networks containing indeterminate primitives. The focus of this chapter will be on developing a semantics for indeterminate networks that, like Kahn's, is fully abstract. We will do this without putting any restriction on the type of indeterminacy allowed; neither restricting ourselves to networks that must contain at least a particular level, nor to networks that contain at most a particular level.

We begin by motivating and describing an approach of the first type – a semantics that is fully abstract for networks that contain fair merge. We will go on to extend this to a model that is fully abstract for all networks. The drawback, as we shall see, is a somewhat cumbersome formalism, and the lack of a convenient fixed-point property. In the next chapter we consider a class of networks of the second type mentioned above, and find that by restricting the expressiveness of the indeterminacy allowed, we can achieve both full abstraction and a fixed-point property.

4.1 The Brock-Ackerman Anomaly

In the previous chapter we described Kahn's semantics. This semantics is based on, and coincides with, the (functional) input-output behavior of networks, and is fully abstract largely because the input-output relation is compositional for determinate networks.

Unfortunately, for networks containing certain indeterminate primitives, the input-output relation fails to be compositional, as was first shown by Brock and Ackerman [BA81]. They exhibited two networks that have the same input-output relation, but that are distinguishable in an appropriate context (see Figure 4.1). The operation of the processes in the example networks X and Y is as follows:

D: These processes "double" their input, outputting two copies of every token that appears on their input.

Fair Merge: This is the fair merge described in the previous chapter. It produces on its output a fair interleaving of the tokens that arrive on its inputs; guaranteeing that every input token will eventually appear on the output, and preserving the relative order of input tokens on each channel.

Pass: This simply passes each input token directly to its output.

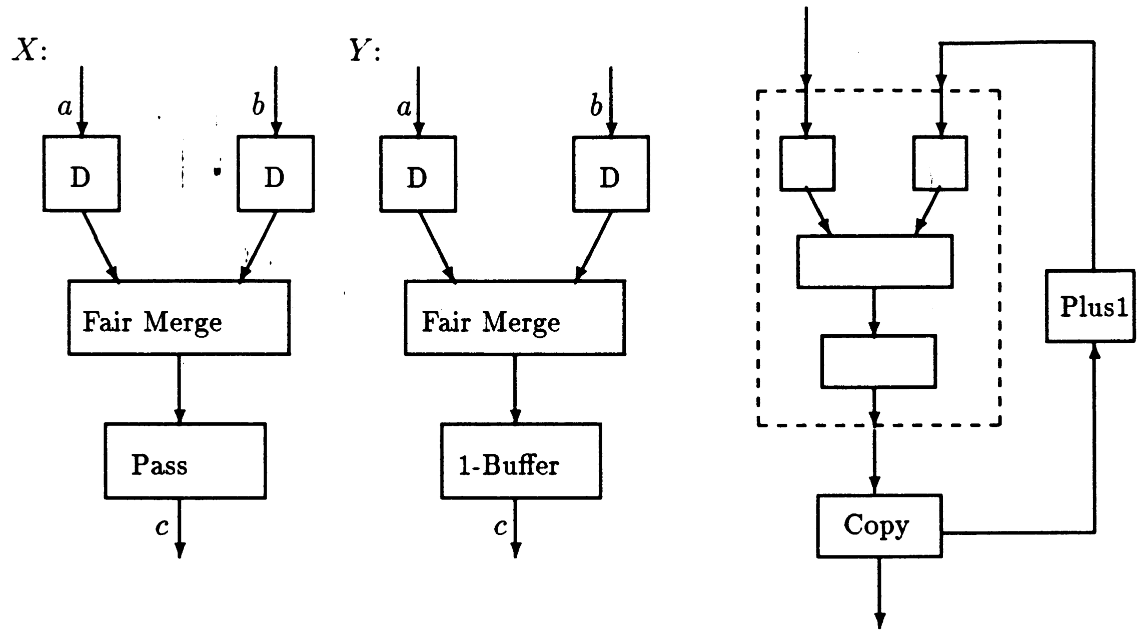

Figure 4.1: The example processes and context of Brock and Ackerman

1-Buffer: This process will buffer its first input token until it receives a second input, then will output both in the order they were received, and then will repeat this behavior. In other words, it will pass its input to its output in pairs, waiting until it has the second of the pair before outputting the first.

It is easy to see that the two networks X and Y have the same input-output relation; since the doubling processes guarantee an even number of tokens on each input to the fair merge, there will be no observable difference if they are output one at a time or two at a time. However, we *can* observe a difference in the behavior of these networks when they are put into the context shown in Figure 4.1. This context consists of a process *Copy* that copies its input to an output channel and also to a feedback loop through a process *Plus1* which adds one to its input.

Consider what happens if we supply the token '5' as input to the composite network with X. This results in two 5's appearing on the left input of the fair merge, so when the first 5 appears on the output of the merge, it may be immediately passed around through the feedback loop, becoming a 6 and then two 6's. The first of these 6's may now be the next token through the merge, since the second 5 on the left input may be arbitrarily delayed. In this case, we observe as output of the composite network a stream of tokens beginning $5\,6\,\cdots$.

Now consider what happens if we supply the same input (the token '5') to the composite network with Y. As before, this results in two 5's appearing on the left input of the fair merge, but in this case when the first 5 appears on the output the *1-Buffer* process holds onto it until the second 5 appears. Hence, the only output we may observe from this network is a stream of tokens beginning $5\,5\,\cdots$.

Thus, we see that the two networks with the same input-output behavior are in fact distinguishable, and hence the straightforward generalization of Kahn's principle based on the input-output relation fails even to be compositional. Clearly, we must look for a semantic representation that captures more information than the input-output relation in order to allow us to distinguish such networks and achieve compositionality.

4.2 Semantics for Fair Merge Networks

In the previous chapter we defined the *trace* of a network execution. In this section we describe a semantic model for indeterminate networks based on traces, and present the result (originally due to Jonsson [Jon89]) that such a model is fully abstract for the class networks containing fair merge.

Let $\mathcal{T}[\![N]\!]$ be the set of all possible traces of the network N. It is clear from the definitions that $\mathcal{T}[\![N]\!]$ determines the input-output relation $\mathcal{IO}[\![N]\!]$, since the input-output behavior is just the histories of the input and output channels of a trace. The representation of networks by sets of traces is a more detailed model than the input-output relation because of the relative timing information present in traces. The Brock-Ackerman example points out that we need a more detailed model, and in fact we see that the model $\mathcal{T}[\![\cdot]\!]$ does distinguish the processes in the previous section.

Example 4.1. Notice that the trace

$$\langle a,5\rangle\langle c,5\rangle\langle c,5\rangle\langle b,6\rangle\cdots$$

is a possible trace of either network X or Y of the previous section, corresponding to the case where both 5's on the left input to the fair merge are output before anything on the right input. However, the trace

$$\langle a,5\rangle\langle c,5\rangle\langle b,6\rangle\langle c,6\rangle\cdots$$

is a possible trace of network X, but not of network Y. If for network Y the only input event was a 5 on channel a, a 5 could only be produced on the output channel c if the buffer process had already seen both the duplicated 5's. In this case, the second output event must also be a 5, which is not true of this trace. Hence, $\mathcal{T}[\![X]\!] \neq \mathcal{T}[\![Y]\!]$. ∎

More generally, the trace model reclaims the property of compositionality that the input-output relation loses for indeterminate dataflow networks.

Theorem 4.1 (Compositionality) Given an indeterminate dataflow network N' constructed from component networks $N_1, \ldots, N_k$, we can derive the representation $\mathcal{T}[\![N']\!]$ from the representations of the components, $\mathcal{T}[\![N_1]\!], \ldots, \mathcal{T}[\![N_k]\!]$.

Proof: The proof proceeds by describing the operations of network composition for trace sets, and showing that they correspond to the operational definitions. Given $\mathcal{T}[\![N]\!]$ and $\mathcal{T}[\![M]\!]$, we have

$$\mathcal{T}[\![N||M]\!] = \{T'' | T'' \text{ is a shuffle of some } T \in \mathcal{T}[\![N]\!] \text{ and } T' \in \mathcal{T}[\![M]\!]\}.$$

Given $\mathcal{T}[\![N]\!]$, we have

$\mathcal{T}[\![\mathsf{loop}(a, b, N)]\!] =$

$\{T| \ \exists T' \in \mathcal{T}[\![N]\!]$ such that:

$H_a(T') = H_b(T')$, and

for all i, the ith output event on channel b in T' preceeds

the ith input event on channel a in T', and

T is equal to T' with the events on channels a and b deleted.$\}$

It is intuitively clear that these operations on trace sets correspond to the operational definitions of aggregation and looping, but the proofs are quite intricate, so they are omitted here. We refer the reader to Jonsson [Jon89] for a detailed proof of the compositionality of traces. ∎

The compositionality of traces leads directly to a stronger property – that if two networks are identified in the trace model, then they are observationally indistinguishable in all network contexts. In other words, that $\mathcal{T}[\![\cdot]\!]$ is an adequate model.

Theorem 4.2 (Adequacy) $\mathcal{T}[\![\cdot]\!]$ is adequate for indeterminate dataflow nets. That is, given networks N and M, if $\mathcal{T}[\![N]\!] = \mathcal{T}[\![M]\!]$ then $\mathcal{IO}[\![\mathcal{C}[N]]\!] = \mathcal{IO}[\![\mathcal{C}[M]]\!]$ for all network contexts $\mathcal{C}[\,]$.

Proof: If $\mathcal{T}[\![N]\!] = \mathcal{T}[\![M]\!]$, then by compositionality we know that $\mathcal{T}[\![\mathcal{C}[N]]\!] = \mathcal{T}[\![\mathcal{C}[M]]\!]$ for all contexts $\mathcal{C}[\,]$, and since we know that $\mathcal{T}[\![\,]\!]$ determines $\mathcal{IO}[\![\,]\!]$, this implies $\mathcal{IO}[\![\mathcal{C}[N]]\!] = \mathcal{IO}[\![\mathcal{C}[M]]\!]$ for all contexts $\mathcal{C}[\,]$. ∎

Actually, it is not altogether surprising that the trace model is adequate, given the amount of extra relative timing information a trace contains over the input-output relation. However, the amount of timing information added by $\mathcal{T}[\![\cdot]\!]$ is not as much as it may at first appear, as is shown in the following theorem, since we represent a network by the set of all its possible traces.

Theorem 4.3. If $T \in \mathcal{T}[\![N]\!]$ and T' satisfies the following two properties:

1) $H_c(t) = H_c(t')$ for all channels c;

2) Given input channel c_{in}, output channel c_{out}, and integers i, j, if the ith input event on channel c_{in} precedes the jth output event on channel c_{out} in T, then the same is true in T';

then T' is also in $\mathcal{T}[\![N]\!]$.

Proof: See Jonsson and Kok [JK88] or Shanbhogue [Sha90]. ∎

In other words, $\mathcal{T}[\![N]\!]$ is closed under the delaying of output events and the exchanging of adjacent input events or output events on different channels, subject to the condition that the overall input-output behavior is not changed. In the next chapter we will discuss another representation that avoids some of these redundancies of trace sets.

The importance of this closure property is that if a model is too detailed it may fail to be fully abstract by distinguishing networks that are observably equivalent. Jonsson showed that the model $\mathcal{T}[\![\cdot]\!]$ is in fact fully abstract for networks containing fair merge.

Theorem 4.4 (Full Abstraction for Fair Merge Networks) Given N and M in the class of indeterminate dataflow networks using fair merge, $\mathcal{T}[\![N]\!] = \mathcal{T}[\![M]\!]$ if and only if $\mathcal{IO}[\![\mathcal{C}[N]]\!] = \mathcal{IO}[\![\mathcal{C}[M]]\!]$ for all contexts $\mathcal{C}[\,]$.

Proof: The 'only if' direction is the adequacy of Theorem 4.2. We prove the other direction by showing that if $\mathcal{T}[\![N]\!] \neq \mathcal{T}[\![M]\!]$ then there exists a context $\mathcal{C}[\,]$ such that $\mathcal{IO}[\![\mathcal{C}[N]]\!] \neq \mathcal{IO}[\![\mathcal{C}[M]]\!]$. The context we exhibit is independent of N and M, and is illustrated in Figure 4.2. It is closely related to one described by Kok [Kok88]. The operation of the processes of $\mathcal{C}[\,]$ are as follows:

Fair Merge: This is as described before.

in-tag$_j$: These "tag" the input channels. For each token d that *in-tag$_j$* gets as input, it builds and outputs the "tagged" token $\langle in_j, d\rangle$.

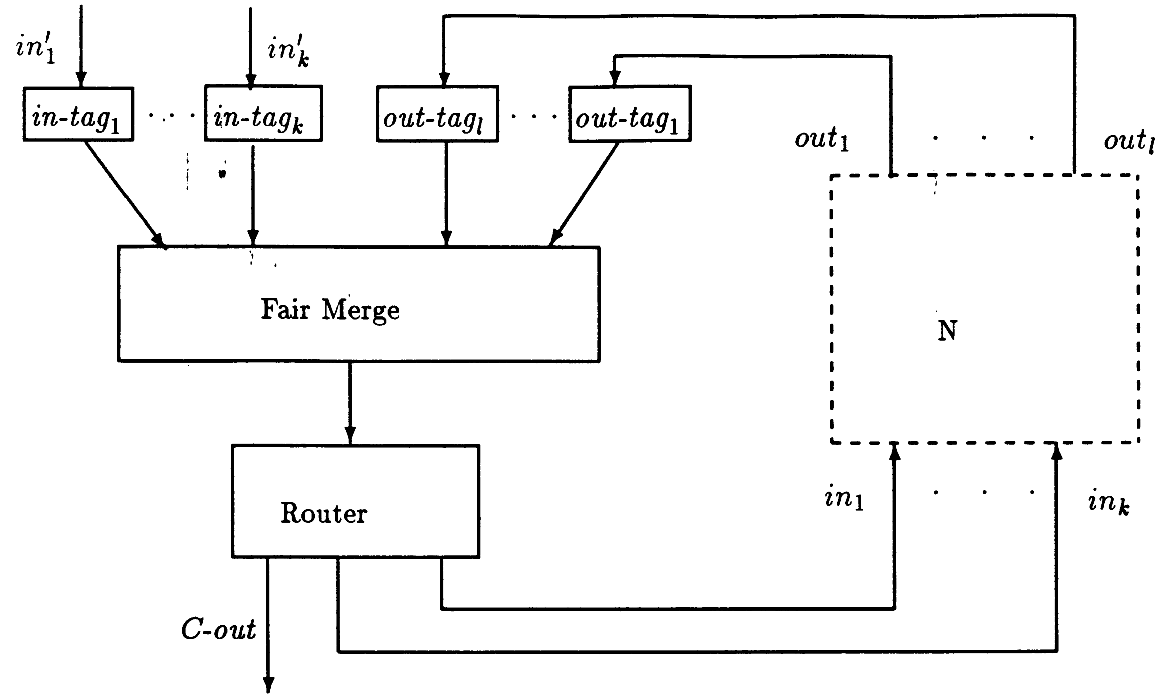

Figure 4.2: The distinguishing context $\mathcal{C}[\,]$

out-tag$_j$: These "tag" the output channels of N. For each token d that *out-tag*$_j$ gets as input, it builds and outputs the "tagged" token $\langle out_j, d\rangle$.

Router: This "directs the traffic" coming out of the merge. For each token it gets as input, it immediately copies it to channel *C-out*. Additionally, if the token is of the form $\langle in_j, d\rangle$ then the Router process will output the token d on channel in_j.

N : This can be any network with input channels $I_N = \{in_1, \ldots, in_k\}$ and output channels $O_N = \{out_1, \ldots, out_l\}$.

The proof rests on showing that the composite network $\mathcal{C}[N]$ has the property that it generates $\mathcal{T}[\![N]\!]$, the traces of the network N. In other words, we must show that T is a possible output of $\mathcal{C}[N]$ on input i if and only if T is a possible trace of the network N and T has input history i. We sketch this proof here, and again refer the reader to Jonsson [Jon89] for details.

If $T \in \mathcal{T}[\![N]\!]$, we see that T is a possible output of $\mathcal{C}[N]$ by noting that if N behaves according to T, then the tagged input and output tokens may arrive at the inputs to the fair merge in the same order as they appear in T. Hence, they may be merged in that order, and the output of the composite network will coincide with T.

If $(i, T) \in \mathcal{IO}[\![\mathcal{C}[N]]\!]$ we see that T is a possible trace of N by noting that if N actually behaves according to a trace T', then T must consist of the events of T' in the order they came out of the merge. From the definition of fair merge we know that T will contain all of the events of T', and will preserve the relative order of events on each channel. Hence, from the definition of the context we see that T may differ from T' only in that the output events may be delayed and adjacent input or output events may be exchanged. Since we know from Theorem 4.3 that $\mathcal{T}[\![N]\!]$ is closed under such differences, and that $T' \in \mathcal{T}[\![N]\!]$ by assumption, we conclude that $T \in \mathcal{T}[\![N]\!]$.

This clearly implies full abstraction, for if the observable output of $\mathcal{C}[N]$ is $\mathcal{T}[\![N]\!]$, then $\mathcal{T}[\![N]\!] \neq \mathcal{T}[\![M]\!]$ means that $\mathcal{IO}[\![\mathcal{C}[N]]\!] \neq \mathcal{IO}[\![\mathcal{C}[M]]\!]$. ∎

This seems to completely solve the problem exposed by the Brock-Ackerman anomaly, in that we have a model that is compositional, correctly distinguishes observably different indeterminate networks, and makes no unnecessary distinctions among fair merge networks. However, notice that both the Brock-Ackerman anomaly and the distinguishing context of the previous theorem rely on the presence of the fair merge primitive. We know from the previous chapter that fair merge is only the strongest of many possible indeterminate primitives. Thus, many questions remain unanswered: Do anomalies like that shown by Brock and Ackerman exist at lower levels of the indeterminate hierarchy, or will a semantics like Kahn's based on the input-output relations be fully abstract at some level below fair merge? What kind of semantic models are fully abstract for weaker forms of indeterminacy? Is there a semantic model that is fully abstract for the whole hierarchy?

The answers to these questions are the main results of this chapter.

4.3 An Example for Bounded Indeterminacy

In this section we give an example which shows that the input-output relation of networks is not compositional in the presence of even the weakest form of indeterminacy – bounded choice. This extends the scope of the Brock-Ackerman anomaly to the entire indeterminate hierarchy.

Theorem 4.5. The input-output relation is not compositional for networks containing bounded choice.

Proof: Consider the two network processes each with one input channel and one output channel shown in Figure 4.3. The operation of these processes is as follows: P_1 indeterminately chooses between the two following determinate behaviors:

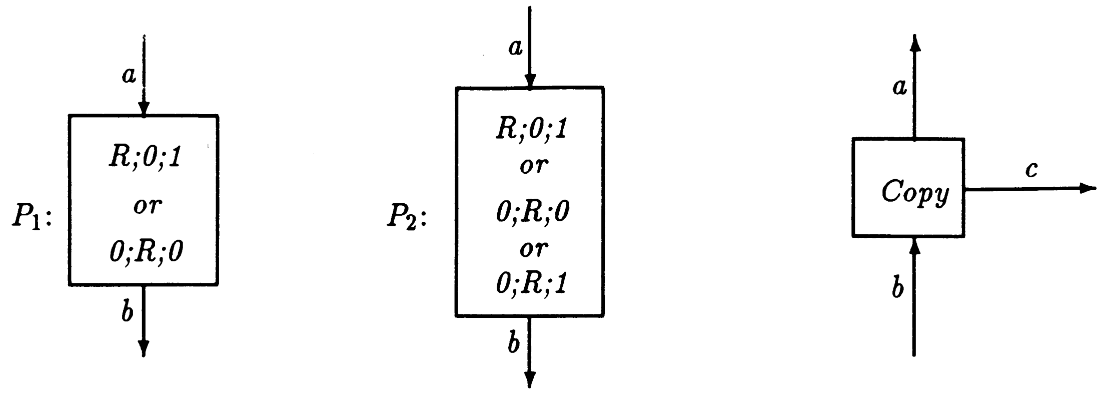

Figure 4.3: The example processes

A) First read a token from channel a, then output a 0 followed by a 1 on channel b, and stop.

B) First output a 0 on b, then read a token from a, then output another 0 on b, and stop.

P_2 chooses among A) and B) above as well as:

C) First output a 0 on b, then read a token from a, then output a 1 on b, and stop.

The above reads are determinate, meaning that they do not time out, and that if channel a is always empty, the process waits forever.

Note that P_1 and P_2 have the same input-output relations, given by $\mathcal{IO}[\![P_1]\!] = \mathcal{IO}[\![P_2]\!] = \{(\epsilon, \epsilon), (\epsilon, 0)\} \cup \{(t, 01), (t, 00) \mid t$ any nonempty stream$\}$.(we use ϵ to denote the empty stream). However, if we look at the networks obtained by composing with the process *Copy*, which copies all tokens from channel b to channels

a and c, we see that they are observably different. Specifically, their input-output relations are given by $\mathcal{IO}[\![Copy[P_1]]\!] = \{(,\epsilon),(,00)\}$ and $\mathcal{IO}[\![Copy[P_2]]\!] = \{(,\epsilon),(,00),(,01)\}$ (since the composite network has no input channels, the input part of the input-output pairs is nonexistent). ∎

Though simple to describe, this is an important example – it shows that we must look for a semantic model more detailed than Kahn's in order to describe networks with any kind of indeterminacy.

While bounded choice may be viewed as the weakest form of purely "internal" indeterminacy, there is a weaker form of "external" indeterminacy. This is defined by a process that has no input channels and a single output channel on which it indeterminately chooses to output nothing or a single token '1'. It is worth noting here that even for networks containing this more restrictive indeterminacy the input-output relation is not compositional, though the example for this case is more complex.

4.4 Full Abstraction for All Indeterminate Networks

In this section we extend the previous work and show that the semantics in which a network N is represented by its set of traces $\mathcal{T}[\![N]\!]$ is fully abstract for all networks, with no restriction on the class of primitives used.

The adequacy of $\mathcal{T}[\![\,]\!]$ described in the prior sections is independent of the class of indeterminacy, so we will show $\mathcal{T}[\![\,]\!]$ to be fully abstract by demonstrating that if $\mathcal{T}[\![N]\!] \neq \mathcal{T}[\![M]\!]$ then there is a network context $\mathcal{C}[\,]$ such that

$\mathcal{IO}[\![\mathcal{C}[N]]\!] \neq \mathcal{IO}[\![\mathcal{C}[M]]\!]$. In fact, we construct such a context that is independent of N and M (except for the number of input and output channels). Furthermore, the context we construct is composed only of determinate (and sequential) processes, and therefore puts no restriction on the class of networks for which $\mathcal{T}[\![\,]\!]$ is fully abstract. We construct this context by borrowing from ideas common in complexity theory and verifying (rather than generating) traces.

Consider the network context $\mathcal{V}[\,]$ shown in Figure 4.4. $\mathcal{V}[\,]$ has $I_{\mathcal{V}[N]} = \{Fin\}$

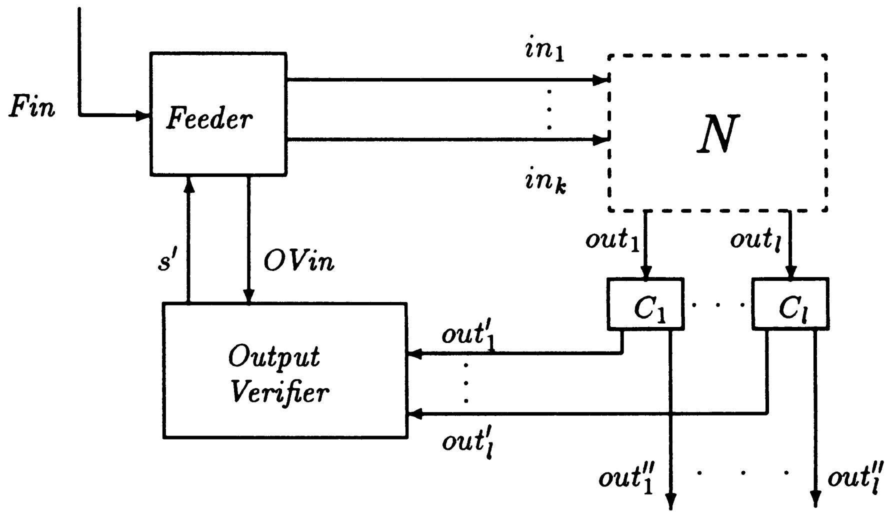

Figure 4.4: The context $\mathcal{V}[\,]$

and $O_{\mathcal{V}[N]} = \{out''_1, \ldots, out''_l\}$, and is composed of the following determinate processes:

Feeder: This expects a trace of N as input on the channel Fin, which it reads one event at a time. For each input event $\langle in_j, d\rangle$ it gets on Fin, it sends the token d out on channel in_j. For each output event $\langle out_j, d\rangle$ it gets on Fin, it sends this pair out on channel $OVin$, and then waits until it receives the distinguished signal token # on channel s' before continuing. This is a determinate process, so if any input it expects is malformed or fails to arrive, it halts. The important aspects of *Feeder* are that it supplies tokens to the channels in_j in the same order as they appear in its input, and its output to the channel $OVin$ is exactly the subsequence of its input consisting of output event pairs.

Output Verifier: This reads a sequence of output events off the channel $OVin$. For each event $\langle out_j, d\rangle$ it receives, it waits until it can read a token d' on channel out'_j, and if $d = d'$, it sends a signal token # out on channel s'. If $d \neq d'$ it halts.

C_j: These are copy nodes. They read from channel out_j and copy onto out'_j, then out''_j.

N : This can be any network with input channels $I_N = \{in_1, \ldots, in_k\}$ and output channels $O_N = \{out_1, \ldots, out_l\}$. N is not required to be determinate.

We now prove that the context $\mathcal{V}[\,]$ actually "verifies" traces of N.

Lemma 4.6. A trace T is a possible trace of the network N if and only if on input T to the network $\mathcal{V}[N]$, a possible output is $H_{out_j}(T)$ on channel out_j for $1 \leq j \leq l$. That is, $T \in \mathcal{T}[\![N]\!] \Longleftrightarrow (T, H_{O_N}(T)) \in \mathcal{IO}[\![\mathcal{V}[N]]\!]$.

Proof: $\Longleftarrow$: We will use the notation T_n to denote the prefix of the trace T consisting of the first n events of T. First, we note the following fact that follows directly from Theorem 4.3: If there exists a trace $S \in \mathcal{T}[\![N]\!]$ such that

(*i*) $\quad H_{I_N}(S) = H_{I_N}(T) \;\;\&\;\; H_{O_N}(S) = H_{O_N}(T)$, and

(*ii*) $\quad \forall n \exists m$ s.t. $H_{I_N}(S_m) = H_{I_N}(T_n) \;\;\&\;\; H_{O_N}(S_m) \sqsupseteq H_{O_N}(T_n)$, then we may conclude $T \in \mathcal{T}[\![N]\!]$, since S may be transformed into it by delaying output events. We will show that under the hypothesis, such a trace S must exist.

Let Γ be a computation of $\mathcal{V}[N]$ such that $H_{I_{\mathcal{V}[N]}}(\Gamma) = T$, and $H_{O_{\mathcal{V}[N]}}(\Gamma) = H_{O_N}(T)$. Since Γ includes a computation of N, we know $\Gamma\lceil_{I_N \cup O_N}$ is in $\mathcal{T}[\![N]\!]$. Let $S = \Gamma\lceil_{I_N \cup O_N}$. It follows from the definition of *Feeder* that $\Gamma\lceil_{I_N} = T\lceil_{I_N}$, and hence $H_{I_N}(S) = H_{I_N}(T)$. By assumption, we have $H_{\{out''_1,\ldots,out''_l\}}(\Gamma) = H_{O_N}(T)$, so from the definition of the C_i, we know that $H_{O_N}(\Gamma) = H_{O_N}(S) = H_{O_N}(T)$.

Now, for all n we define Γ_n as the largest prefix of Γ such that $\Gamma_n\lceil_{I_N} = T_n\lceil_{I_N}$, and $H_{OVin}(\Gamma_n)\lceil_{O_N} = T_n\lceil_{O_N}$. Let $S_{m(n)} = \Gamma_n\lceil_{I_N \cup O_N}$. Then we see that

$$\begin{aligned} \Gamma_n\lceil_{I_N} = T_n\lceil_{I_N} \quad &\Rightarrow \quad H_{I_N}(\Gamma_n) = H_{I_N}(T_n) \\ &\Rightarrow \quad H_{I_N}(S_{m(n)}) = H_{I_N}(T_n). \end{aligned}$$

From the definition of *Output Verifier* and the maximality of Γ_n, we know that every output event on $OVin$ is matched by an event on an out' channel. Hence, for $1 \leq j \leq l$,

$$\begin{aligned} H_{OVin}(\Gamma_n)\lceil_{out_j} \sqsubseteq \Gamma_n\lceil_{out'_j} \quad &\Rightarrow \quad H_{O_N}(H_{OVin}(\Gamma_n)) \sqsubseteq H_{\{out'_1,\ldots,out'_l\}}(\Gamma_n) \\ &\Rightarrow \quad H_{O_N}(T_n) \sqsubseteq H_{\{out'_1,\ldots,out'_l\}}(\Gamma_n) \\ &\qquad\qquad \text{(by the definition of } \textit{Feeder}\text{)} \\ &\Rightarrow \quad H_{O_N}(T_n) \sqsubseteq H_{O_N}(\Gamma_n) = H_{O_N}(S_{m(n)}) \\ &\qquad\qquad \text{(by the definition of } C_i\text{)} \end{aligned}$$

Now, by applying the fact stated above, we conclude that $T \in \mathcal{T}[\![N]\!]$.

$\Longrightarrow$: Assume $T \in \mathcal{T}[\![N]\!]$. We show how to construct compatible traces of the components of $\mathcal{V}[N]$, such that their composition, T^* (which is a trace of $\mathcal{V}[N]$), generates the input-output pair

$$(H_{I_{\mathcal{V}[N]}}(T^*), H_{O_{\mathcal{V}[N]}}(T^*)) = (T, H_{O_N}(T)).$$

We construct the traces for each component based on T as follows:

Feeder: For each event $\langle in_j, d\rangle$ in T, replace it with the two events $\langle Fin, \langle in_j, d\rangle\rangle\langle in_j, d\rangle$. Replace each event $\langle out_j, d\rangle$ with the events $\langle Fin, \langle out_j, d\rangle\rangle\langle OVin, \langle out_j, d\rangle\rangle\langle s', \#\rangle$.

Output Verifier: Leave out all events in T of the form $\langle in_j, d\rangle$, and replace each event $\langle out_j, d\rangle$ by the events $\langle OVin, \langle out_j, d\rangle\rangle\langle out'_j, d\rangle\langle s', \#\rangle$.

C_i: For each event of the form $\langle out_i, d\rangle$ in T, replace it by the events $\langle out_i, d\rangle\langle out'_i, d\rangle\langle out''_i, d\rangle$. Leave out all other events in T.

It is straightforward to see that the composition of the traces constructed above and T yields the trace T^* defined as follows: For each event $\langle in_j, d\rangle$ in T, replace it with $\langle Fin, \langle in_j, d\rangle\rangle$, and replace each event $\langle out_j, d\rangle$ with the events $\langle Fin, \langle out_j, d\rangle\rangle\langle out''_j, d\rangle$. Clearly, T^* is in $\mathcal{T}[\![\mathcal{V}[N]]\!]$, $H_{I_{\mathcal{V}[N]}}(T^*) = T$, and $H_{O_{\mathcal{V}[N]}}(T^*) = H_{O_N}(T)$. Hence, $(T, H_{O_N}(T)) \in \mathcal{IO}[\![\mathcal{V}[N]]\!]$. ∎

From the above lemma, the full abstraction of $\mathcal{T}[\![\,]\!]$ follows directly.

Theorem 4.7 (Full Abstraction) Given dataflow networks N and M, $\mathcal{T}[\![N]\!] = \mathcal{T}[\![M]\!]$ if and only if $\mathcal{IO}[\![\mathcal{C}[N]]\!] = \mathcal{IO}[\![\mathcal{C}[M]]\!]$ for all network contexts $\mathcal{C}[\,]$.

Proof: We already know that $\mathcal{T}[\![\,]\!]$ is adequate – this is the 'only if' direction. For the other direction, assume there exists $T \in \mathcal{T}[\![N]\!], T \notin \mathcal{T}[\![M]\!]$. Then by the above lemma, the input-output pair $(T, H_{O_N}(T))$ is in $\mathcal{IO}[\![\mathcal{V}[N]]\!]$ but not $\mathcal{IO}[\![\mathcal{V}[M]]\!]$, so $\mathcal{IO}[\![\mathcal{V}[N]]\!] \neq \mathcal{IO}[\![\mathcal{V}[M]]\!]$. ∎

This completely solves the problem of finding a fully abstract semantics for the full range of indeterminate primitives. Thus, we have succeeded in generalizing this aspect of Kahn's semantics. However, the trace model is cumbersome to work with and does not have a nice fixed-point principle like Kahn's semantics. In the next chapter we take a somewhat different approach to find a model that is both fully abstract and has a convenient fixed-point principle.

Chapter 5

On Oraclizable Networks and Kahn's Principle

In this chapter we generalize another aspect of Kahn's principle. The approach we take here is to concentrate our attention on a particular type of indeterminacy, and to consider a class of indeterminate networks called *oraclizable* networks. This class represents the situation where indeterminacy results from imperfect knowledge of the behavior of a system, and is modeled as determinate network that has an extra input channel hidden from the external observer. Since we do not know the contents of the hidden input, it acts as the source of indeterminacy in the network. We develop a particularly nice model for the class of oraclizable networks, and the model that we develop is a true generalization of Kahn's – it is fully abstract and has the same fixed-point property.

In the first section we begin by defining a useful alternative to the trace model for representing networks. In the second section we give a formal definition of oraclizable networks and present our first semantic model for this class. This model is a straightforward extension of Kahn's in which oraclizable networks are represented by *sets* of stream-valued functions, corresponding to all their possible determinate behaviors. With this representation, the fixed-point property of Kahn's semantics extends directly to this model by applying it to each of the possible equations.

Unfortunately, this straightforward representation fails to be fully abstract. In the third section we modify this model and represent oraclizable networks by sets of functions that are closed in a certain sense. With this modification, we show that this second semantics becomes fully abstract, and preserves the fixed-point property of the first.

5.1 Checkpoint Sequences

In this section we describe an alternative representation for indeterminate processes, which we call *checkpoint sequences.* Checkpoint sequences are defined in terms of traces, and are equivalent in expressive power, but they have two important advantages: First, they more closely resemble descriptions of stream-valued functions than traces; and second, they avoid some of the redundancy associated with the ordering of unrelated input (or output) events necessary in traces.

Intuitively, a checkpoint of a network N is a pair of finite streams (i, o), such that on input i, N may be observed to produce output at least o (if N has more than one input (or output) channel, then i (or o) is an appropriate channel indexed tuple of finite streams). The finiteness of i and o guarantee that this observation can be made in finite time.

A checkpoint sequence is a sequence of checkpoints

$$(i_0, o_0), (i_1, o_1), \ldots (i, o)$$

where $i_n \sqsubseteq i_{n+1}, o_n \sqsubseteq o_{n+1}\ \forall n$, and $\sqcup_n(i_n, o_n) = (i, o)$ (i and o need not be finite in the limit). Such a checkpoint sequence is meant to represent a computation of network N during which on input i_0, at least o_0 is observed as output, and after this the input is increased to i_1, and at least o_1 is observed, and then the input is increased to i_2, etc. The total input to N is then i, and the total output is o.

Formally, we define checkpoint sequences in terms of traces.

Definition 5.1. A *checkpoint sequence* of network M

$$(i_0, o_0), (i_1, o_1), \ldots (i, o)$$

is shorthand for a trace T of M consisting of all the input events of i_0, followed by the output events of o_0, followed by the input events of $i_1 - i_0$, followed by the output events of $o_1 - o_0$, etc. Furthermore, the entire input history of T must be i, and the output history of T must be o. Note that a given checkpoint sequence actually represents a family of traces related by the permutation of adjacent input (or output) events on different channels, since trace sets are closed under such permutations.

Given a network M we define $\mathcal{CK}[\![M]\!]$ to be the set of all checkpoint sequences of the network M. It is clear from the definition that $\mathcal{CK}[\![M]\!]$ determines, and is determined by, $\mathcal{T}[\![M]\!]$, and therefore for all networks M and N, $\mathcal{CK}[\![M]\!] = \mathcal{CK}[\![N]\!]$ if and only if $\mathcal{T}[\![M]\!] = \mathcal{T}[\![N]\!]$. Thus, $\mathcal{CK}[\![\,]\!]$ is automatically a fully abstract model for all indeterminate dataflow networks.

By grouping the inputs and outputs together as in checkpoint sequences, we obtain a representation that looks like a function specification, as opposed to the sequence of operational steps of a trace. This will be very useful in the following sections.

5.2 The Direct Semantics of Oraclizable Networks

Definition 5.2. An indeterminate network M is *oraclizable* if it is operationally equal to an indeterminate network M_O without input channels (the "oracle"), connected to some inputs of a determinate network M_D.

Given an oraclizable network M, we can identify its oracle part M_O with the set of possible outputs of M_O. Additionally, we can regard its determinate part M_D, which we know acts as a function f_{M_D} from oracle inputs and external inputs to outputs, as function from oracle inputs to functions from external inputs to outputs. With this view it is easy to see that the class of oraclizable networks is closed under composition, since we can compose oracles simply by combining the sets of possible outputs. This view also leads naturally to our first semantic model.

Definition 5.3. Given an oraclizable network M, we represent it by its set of possible functional behaviors

$$\mathcal{F}_1[\![M]\!] \stackrel{\text{def}}{=} \bigcup_{o \in M_O} f_{M_D}(o).$$

Lemma 5.1.

$$\mathcal{F}_1[\![M||N]\!] = \{g | g = \langle f, f' \rangle, f \in \mathcal{F}_1[\![M]\!], f' \in \mathcal{F}_1[\![N]\!]\}$$
$$\mathcal{F}_1[\![\mathsf{loop}(a, b, M)]\!] = \{f | f = \mathsf{fix}(a, b, f'), f' \in \mathcal{F}_1[\![M]\!]\}$$

Proof: These follow directly from the definitions. ∎

$\mathcal{F}_1[\![\,]\!]$ is an attractive semantic model, since the network composition operations of aggregation and looping correspond to function aggregation and least fixed point applied to each of the functions in the representation, as we desire. Now we compare the semantics $\mathcal{F}_1[\![\,]\!]$ to the fully abstract checkpoint sequence semantics $\mathcal{CK}[\![\,]\!]$.

Lemma 5.2.

$$\mathcal{CK}[\![M]\!] = \{(i_0, o_0), (i_1, o_1), \ldots (i, o) | \exists f \in \mathcal{F}_1[\![M]\!], f(i) = o, f(i_n) \sqsupseteq o_n \text{ for all } n\}$$

Proof: Direct from the definition of $\mathcal{F}_1[\![M]\!]$ and the observation that for a determinate network D,

$$\mathcal{CK}[\![D]\!] = \{(i_0, o_0), (i_1, o_1), \ldots (i, o) | f_D(i) = o, f_D(i_n) \sqsupseteq o_n \text{ for all } n\}$$

∎

From Lemma 5.2 we conclude that $\mathcal{F}_1[\![M]\!] = \mathcal{F}_1[\![N]\!]$ implies $\mathcal{CK}[\![M]\!] = \mathcal{CK}[\![N]\!]$, and hence $\mathcal{F}_1[\![\,]\!]$ is adequate. It is not, however, fully abstract, as is shown in the following example.

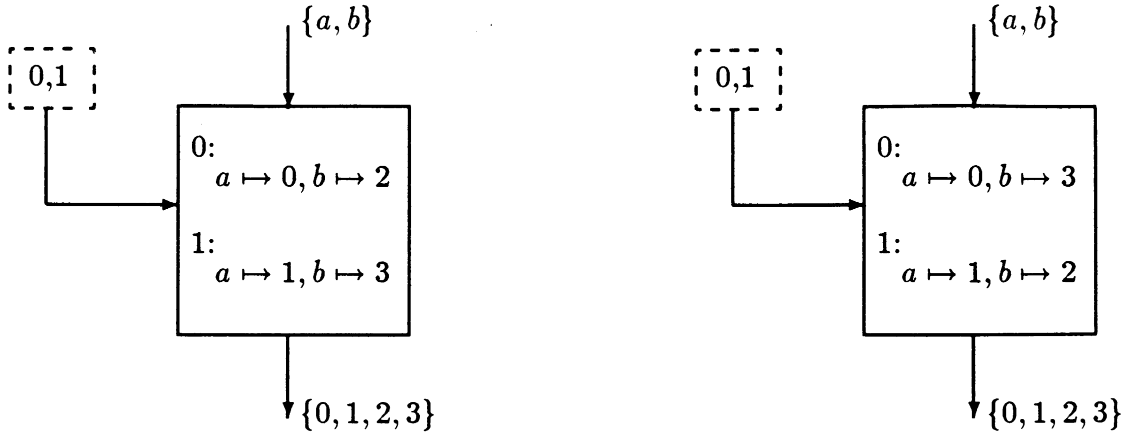
$\{a, b\}$
0,1
0:
$a \mapsto 0, b \mapsto 2$
1:
$a \mapsto 1, b \mapsto 3$
$\{0,1,2,3\}$
$\{a, b\}$
0,1
0:
$a \mapsto 0, b \mapsto 3$
1:
$a \mapsto 1, b \mapsto 2$
$\{0,1,2,3\}$

Figure 5.1: Two indistinguishable processes with different sets of functions.

Example 5.1. Consider the processes shown in Figure 5.1. Both of the processes take as input a single token, either a or b, and if it is an a produce either 0 or 1, and if it is a b produce either 2 or 3. However, the process on the left uses its oracle to choose between the functions $a \mapsto 0, b \mapsto 2$ and $a \mapsto 1, b \mapsto 3$, while the process on the right uses its oracle to choose between the two different functions $a \mapsto 0, b \mapsto 3$ and $a \mapsto 1, b \mapsto 2$. Thus, these two processes are distinguished by the semantic model $\mathcal{F}_1[\![\,]\!]$, even though they are observably equivalent. Hence, $\mathcal{F}_1[\![\,]\!]$ is not fully abstract. ∎

5.3 A Fully Abstract Variation

Intuitively, $\mathcal{F}_1[\![M]\!]$ fails to be fully abstract because it only includes the functional behaviors explicit in M, while there may be other functions inherent in the behavior of M, though not corresponding to any single oracle value. In order to be fully abstract, the representation must identify networks differing only by such functions.

Definition 5.4. Given a set of functions F, $Cl(F)$ is the closure of F under the addition of functions not finitely distinguishable from F. These are the functions inherent in the behavior of F.

$$\begin{aligned} Cl(F) \overset{\text{def}}{=} \{f \mid\ & \forall \text{ chains of finite inputs } i_0 \sqsubseteq i_1 \sqsubseteq \cdots, i = \bigsqcup i_n, \text{ and} \\ & \forall \text{ chains of finite outputs } o_0 \sqsubseteq o_1 \sqsubseteq \cdots, f(i) = \bigsqcup o_n, \\ & o_n \sqsubseteq f(i_n) \text{ for each } n, \\ & \exists f' \in F \text{ with } f'(i) = f(i), f'(i_n) \sqsupseteq o_n \text{ for each } n\} \end{aligned}$$

Definition 5.5. Given an oraclizable network M, we represent it by the closure of its set of functional behaviors.

$$\mathcal{F}_2[\![M]\!] \overset{\text{def}}{=} Cl(\mathcal{F}_1[\![M]\!])$$

Clearly $\mathcal{F}_2[\![M]\!] \supseteq \mathcal{F}_1[\![M]\!]$ for any M.

As justification for the claim that the additional functions were inherent in M we have the following:

Lemma 5.3. Given an oraclizable network M, suppose M' is an oraclizable network whose functional behaviors are all those in $\mathcal{F}_2[\![M]\!]$ (i.e. $\mathcal{F}_1[\![M']\!] = \mathcal{F}_2[\![M]\!]$). Then M and M' are operationally indistinguishable.

Proof: We will show $\mathcal{CK}[\![M']\!] = \mathcal{CK}[\![M]\!]$. Using Lemma 5.2 and the definition of $\mathcal{F}_2[\![\,]\!]$ we have

$$\mathcal{CK}[\![M']\!] = \{(i_0, o_0), (i_1, o_1), \ldots (i, o) \mid \exists f \in \mathcal{F}_1[\![M']\!], f(i) = o,$$

$$
\begin{aligned}
& f(i_n) \sqsupseteq o_n \text{ for all } n\} \\
= \; & \{(i_0, o_0), (i_1, o_1), \ldots (i, o) | \exists f \in Cl(\mathcal{F}_1[\![M]\!]), f(i) = o, \\
& f(i_n) \sqsupseteq o_n \text{ for all } n\} \\
= \; & \{(i_0, o_0), (i_1, o_1), \ldots (i, o) | \exists f' \in \mathcal{F}_1[\![M]\!], f'(i) = f(i) = o, \\
& f'(i_n) \sqsupseteq o_n \text{ for all } n\} \\
= \; & \mathcal{CK}[\![M]\!]
\end{aligned}
$$

∎

Thus, the sets $\mathcal{F}_2[\![M]\!]$ and $\mathcal{F}_1[\![M]\!]$ represent the same operational behavior. However, $\mathcal{F}_2[\![\,]\!]$ is general enough that it is fully abstract.

Theorem 5.4 (Full Abstraction for Oraclizable Networks) Given oraclizable networks M and N,

$$\mathcal{F}_2[\![M]\!] = \mathcal{F}_2[\![N]\!] \iff \mathcal{CK}[\![M]\!] = \mathcal{CK}[\![N]\!]$$

Proof: $\Leftarrow$: It is clear from Lemma 5.2 and the definition of $\mathcal{F}_2[\![\,]\!]$ that

$$
\begin{aligned}
\mathcal{F}_2[\![M]\!] = \{f | \; & \forall \text{ chains } i_0 \sqsubseteq i_1 \sqsubseteq \cdots, i = \bigsqcup i_n, \text{ and} \\
& \forall \text{ chains } o_0 \sqsubseteq o_1 \sqsubseteq \cdots, f(i) = \bigsqcup o_n, o_n \sqsubseteq f(i_n) \text{ for each } n, \\
& \text{the checkpoint sequence } (i_0, o_0), (i_1, o_1), \ldots (i, f(i)) \text{ is in } \mathcal{CK}[\![M]\!]\}
\end{aligned}
$$

Hence $\mathcal{CK}[\![M]\!]$ determines $\mathcal{F}_2[\![M]\!]$, and the result follows.

$\Rightarrow$: Say $(i_0, o_0), (i_1, o_1), \ldots (i, o) \in \mathcal{CK}[\![M]\!]$. Then by Lemma 5.2 we know there exists $f \in \mathcal{F}_1[\![M]\!]$ with $f(i) = o, f(i_n) \sqsupseteq o_n$ for each n. But $f \in \mathcal{F}_1[\![M]\!] \subseteq \mathcal{F}_2[\![M]\!] = \mathcal{F}_2[\![N]\!]$, so from the above equality we conclude $(i_0, o_0), (i_1, o_1), \ldots (i, o) \in \mathcal{CK}[\![N]\!]$.

∎

Thus, $\mathcal{F}_2[\![\,]\!]$ is a modification of $\mathcal{F}_1[\![\,]\!]$ that is fully abstract. We now verify that $\mathcal{F}_2[\![\,]\!]$ preserves the desirable composition and fixed-point properties of $\mathcal{F}_1[\![\,]\!]$.

Theorem 5.5. $\mathcal{F}_2[\![\,]\!]$ has the same composition and fixed-point properties as $\mathcal{F}_1[\![\,]\!]$. Specifically,

$$\mathcal{F}_2[\![M||N]\!] = \{g | g = \langle f, f' \rangle, f \in \mathcal{F}_1[\![M]\!], f' \in \mathcal{F}_2[\![N]\!]\}$$

$$\mathcal{F}_2[\![\mathsf{loop}(a,b,M)]\!] = Cl(\{g | g = \mathsf{fix}(a,b,g'), g' \in \mathcal{F}_2[\![M]\!]\}).$$

Proof: The aggregation property follows directly from the definitions.
We see $\mathcal{F}_2[\![\mathsf{loop}(a,b,M)]\!] \subseteq Cl(\{g | g = \mathsf{fix}(a,b,g'), g' \in \mathcal{F}_2[\![M]\!]\})$ directly, since

$$\mathcal{F}_2[\![\mathsf{loop}(a,b,M)]\!] = Cl(\mathcal{F}_1[\![\mathsf{loop}(a,b,M)]\!]) = Cl(\{f | f = \mathsf{fix}(a,b,f'), f' \in \mathcal{F}_1[\![M]\!]\})$$

and $\mathcal{F}_1[\![M]\!] \subseteq \mathcal{F}_2[\![M]\!]$.
Finally, we show $\mathcal{F}_2[\![\mathsf{loop}(a,b,M)]\!] \supseteq \{g | g = \mathsf{fix}(a,b,g'), g' \in \mathcal{F}_2[\![M]\!]\}$ via the following somewhat intricate construction.

Given $g = \mathsf{fix}(a,b,g')$ for some $g' \in \mathcal{F}_2[\![M]\!]$, choose chains

$$i_0 \sqsubseteq i_1 \sqsubseteq \cdots i, \quad \text{and}$$

$$o_0 \sqsubseteq o_1 \sqsubseteq \cdots o,$$

with $g(i) = o$, and $g(i_n) \sqsupseteq o_n$ for all n. For every n we define $p_n = g(i_n)$, which we know means that p_n is the least tuple of output streams such that there exists a stream l_n (the 'looped' input) with $g'(i_n, l_n) = (p_n, l_n)$ (where l_n is actually the ath component of the input and the bth component of the output). Note that $(l_n)_{n \geq 0}$ and $(p_n)_{n \geq 0}$ form increasing chains (not necessarily of finites), and for $l = \bigsqcup_n l_n$ and $p = \bigsqcup_n p_n$ we have $g'(i,l) = (p,l)$ (and thus $g(i) = p = o$).

We now introduce the notation $[s]_n$ to denote the prefix of the stream s of length at most n. Similarly, if s is a tuple of streams, $[s]_n$ is defined componentwise. Using this notation, for each n we define chains of finites $(l_n^m)_{n\geq 0}$ and $(p_n^m)_{n\geq 0}$ approximating l_n and p_n as follows:

$$\begin{aligned} &l_n^0 \overset{\text{def}}{=} \epsilon \\ &(p_n^m, l_n^{m+1}) \overset{\text{def}}{=} [g'(i_n, l_n^m)]_m \quad \forall m \geq 0. \end{aligned}$$

Since l_n, p_n are least fixed points, we know from this construction that $\bigsqcup_m l_n^m = l_n$ and $\bigsqcup_m p_n^m = p_n$ for all n. It is also easy to see that for $n \leq n'$ and $m \leq m'$ we have

$$l_n^m \sqsubseteq l_{n'}^{m'} \quad \text{and} \quad p_n^m \sqsubseteq p_{n'}^{m'}.$$

For every n we have $p_n = g(i_n) \sqsupseteq o_n$, therefore we know that $p_n^m \sqsupseteq o_n$, for some finite m. We use this fact to define a family of indices k_n as follows:

$$\begin{aligned} k_0 &= \text{the least index such that } p_0^{k_0} \sqsupseteq o_0 \\ k_{n+1}\ (n \geq 0) &= \text{the greater of } k_n + 1, \text{ and} \\ &\qquad\qquad \text{the least index } k_{n+1} \text{ such that } p_{n+1}^{k_{n+1}} \sqsupseteq o_{n+1}. \end{aligned}$$

Finally, we may consider the following chains of finites:

$$\begin{aligned} &(i_0, l_0^{k_0}) \sqsubseteq (i_1, l_1^{k_1}) \sqsubseteq (i_2, l_2^{k_2}) \sqsubseteq \cdots (i, l) \\ &(p_0^{k_0}, l_0^{k_0+1}) \sqsubseteq (p_1^{k_1}, l_1^{k_1+1}) \sqsubseteq (p_2^{k_2}, l_2^{k_2+1}) \sqsubseteq \cdots (p, l) \end{aligned}$$

with $g'(i_n, l_n^{k_n}) \sqsupseteq (p_n^{k_n}, l_n^{k_n+1}) \forall n$, $g'(i, l) = (p, l)$, and $p_n^{k_n} \sqsupseteq o_n \forall n$, $p = o$. Then, since $g' \in \mathcal{F}_2[\![M]\!]$, there exists $f' \in \mathcal{F}_1[\![M]\!]$ with $f'(i_n, l_n^{k_n}) \sqsupseteq (p_n^{k_n}, l_n^{k_n+1}) \forall n$, and $f'(i, l) = (p, l)$. Now let $f = \mathsf{fix}(a, b, f') \in \mathcal{F}_1[\![\mathsf{loop}(a, b, M)]\!]$, and we have

$f(i_n) \sqsupseteq o_n \forall n$, $f(i) = p = o$. Hence, we may conclude $g \in Cl(\mathcal{F}_1[\![\mathsf{loop}(a, b, M)]\!]) = \mathcal{F}_2[\![\mathsf{loop}(a, b, M)]\!]$. ∎

Thus, we have succeeded in finding what we want: A semantic model for the class of oraclizable networks that is fully abstract and that comes equipped with a fixed-point principle, thereby generalizing both aspects of Kahn's principle to this class. In the next chapter we use this characterization to investigate the relative expressiveness of this and other classes of indeterminate dataflow networks.

Chapter 6

Expressiveness of Different Classes

In this chapter we use the characterization of the previous chapter to explore the relationship of the class of oraclizable dataflow networks with other classes of indeterminate dataflow networks. We will show that the oraclizable networks are closely related to, but different from, both the infinity-fair merge networks and the Egli-Milner monotone networks. In doing so, we will discover new relations between the specification of a network in terms of its input-output relation and its possible implementation in terms of the various indeterminate primitives.

6.1 The Universality of Oraclizable Networks

In the previous chapter we restricted ourselves to the class of oraclizable networks and discovered that the representation by certain sets of functions was fully abstract. This representation is a particularly pleasing one, and we naturally want to know for what other classes of networks might it apply. In this section we answer that question, and show that the oraclizable networks are universal for this representation; i.e. that the oraclizable networks are the largest class of networks representable by sets of functions.

Theorem 6.1. The oraclizable networks are exactly those whose behavior can be described by a set of functions.

Proof: By Lemmas 6.2 and 6.3. ∎

Lemma 6.2. The behavior of any oraclizable network is representable by a set of functions.

Proof: This is the semantics of the previous chapter. ∎

Lemma 6.3. Given any set of functions F, there is an oraclizable network M that implements F (i.e. $\mathcal{F}_1[\![M]\!] = F$).

Proof: Given F, we explicitly construct the network M, with determinate part M_D, and oracle part M_O.

The idea behind M_D is that F can be organized into a countably branching tree indexed by infinite integer sequences. As in the previous chapter, for a stream i we use the notation $[i]_n$ to denote the prefix of i of length at most n; similarly for $[f(i)]_n$. Given functions $f, f' \in F$, we write $f \equiv_n f'$ if and only if for all i,

$$[f([i]_n)]_n = [f'([i]_n)]_n.$$

Now we note that there are only countably many equivalence classes of F modulo $\equiv_1$. Hence, we can index them by integers and denote them C_{k_1}. Similarly, for every integer k_1, we index the equivalence classes of C_{k_1} modulo $\equiv_2$ by integers and denote them $C_{k_1 k_2}$. Proceeding in this way, we can define C_s for any finite sequence s of integers. For s infinite, C_s is the intersection of all $C_{s'}$ with s' a prefix of s, and hence is either empty or contains a single function from F. We will let S be the set of infinite sequences s for which C_s is not empty.

Now we define a function P such that given input i and a sequence of integers s, we have

$$P(s,i) \overset{\text{def}}{=} \begin{cases} [f([i]_n)]_n, f \in C_s & \text{if } s \text{ finite of length } n \\ f(i), f \in C_s & \text{if } s \text{ infinite and } C_s \text{ not empty} \\ \bigsqcup\{P(s',i) \mid s' \text{ a finite prefix of } s\} & \text{if } s \text{ infinite and } C_s \text{ empty} \end{cases}$$

It is easy to see that P is continuous, and that for infinite s with C_s not empty, P computes the unique function $f \in F$ indexed by s. We take M_D to be the determinate process that computes P.

Finally, we take M_O to be an oracle process that produces exactly the streams $s \in S$. Clearly, with this definition of M, M can behave like any and all the functions in F – that is, $\mathcal{F}_1[\![M]\!] = F$. Note that although M_D is defined for infinite streams not in S (and may not compute a function in F on such streams), restricting the oracle M_O to the set S assures that the "extra" functions are not possible behaviors for M. ∎

This result is both pleasing, in that it says that there is an exact correspondence between our representation and the class of oraclizable networks, and disappointing, since it means that our representation will not work for any larger class of networks. We will take advantage of the exact correspondence in the following sections, as we compare the oraclizable networks to other classes.

6.2 Oraclizable Networks and Infinity-Fair Merge

In this section we attempt to characterize the class of oraclizable networks in terms of, and relate it to, the various indeterminate network primitives introduced in Chapter 3. Specifically, we relate it to the infinity-fair merge primitive, and show that while any network constructed with infinity-fair merge is oraclizable, the class of infinity-fair merge networks is properly contained in the class of oraclizable networks.

Theorem 6.4. The class of all networks constructible with infinity-fair merge is a proper subclass of the oraclizable networks.

Proof: By the following two lemmas. ∎

Lemma 6.5. All networks constructible with infinity-fair merge and determinate processes are oraclizable.

Proof: Infinity-fair merge is oraclizable, since it is equivalent to an oracle that produces fair bit streams connected to a deterministic merge that uses the oracle input to decide which channel to read next. Since oraclizable networks are closed under composition, the result follows. ∎

Lemma 6.6. The set of oraclizable networks has strictly greater cardinality than the set of infinity-fair merge networks.

Proof: As we have already noted, infinity-fair merge is equivalent to an oracle that produces all fair bit streams connected to a determinate merge. Hence, any infinity-fair merge network can be implemented as this fair oracle connected to some determinate network. Thus, the number of infinity-fair merge networks is bounded by the number of determinate networks, which by Kahn's principle is the same as the number of continuous stream-valued functions. Since the domain of streams is ω-algebraic, this is the same cardinality as the powerset of ω, $\mathcal{P}(\omega)$.

In general, the oracle part of an oraclizable network may emit *any* set of streams. Hence there are at least as many oraclizable networks as the powerset of the domain of streams, $\mathcal{P}(S)$. Since the domain of streams is as large as $\mathcal{P}(\omega)$, the cardinality of the set of oraclizable networks is at least that of $\mathcal{P}(\mathcal{P}(\omega))$, which is strictly greater than the cardinality of $\mathcal{P}(\omega)$. ∎

The proof of Lemma 6.6, while straightforward, is somewhat unsatisfying because of its nonconstructive nature. Of course, because of the difference in cardinalities of the two classes, it is possible to explicitly construct a network that is oraclizable but is not implementable with infinity-fair merge by a diagonalization argument. However, this is a rather contrived example and is nearly as unsatisfying as none at all. Unfortunately, at the present this is the best we can do, since we know of no "natural" example (either in the sense of having a natural description, or in the sense of being implementable with some stronger indeterminate primitive) of an oraclizable network not implementable with infinity-fair merge.

6.3 Oraclizable Networks and Egli-Milner Monotonicity

In Chapter 3, we saw that one of the characteristics that separates the networks built from infinity-fair merge from those at higher levels of the hierarchy is that the infinity-fair merge networks all have input-output relations that are monotone in the Egli-Milner ordering. It had been conjectured that there was an equivalence between the two classes, namely that any Egli-Milner monotone relation was the input-output relation of some oraclizable network. The result of the previous section, coupled with the observation that the input-output relation of any oraclizable network is Egli-Milner monotone, shows that this conjecture fails. However, the apparent lack of a natural oraclizable network not implementable with infinity-fair merge suggests the possibility that the addition of oracles adds only cardinality but no "real" expressiveness. Given this, a modified question suggests itself; perhaps the class of oraclizable networks is equivalent to the Egli-Milner monotone networks.

In this section we consider this question, and show that the two classes are unarguably distinct. We do this by exhibiting a natural (in the sense of being implementable with some indeterminate primitive) Egli-Milner monotone relation that cannot be the input-output relation of any oraclizable network.

Recall that given sets A and B, $A \sqsubseteq_{EM} B$ if and only if

$$\forall a \in A\ \exists b \in B \text{ s.t. } a \sqsubseteq b\ \ \&\ \ \forall b \in B\ \exists a \in A \text{ s.t. } a \sqsubseteq b.$$

We say that the input-output relation of a network M is Egli-Milner monotone

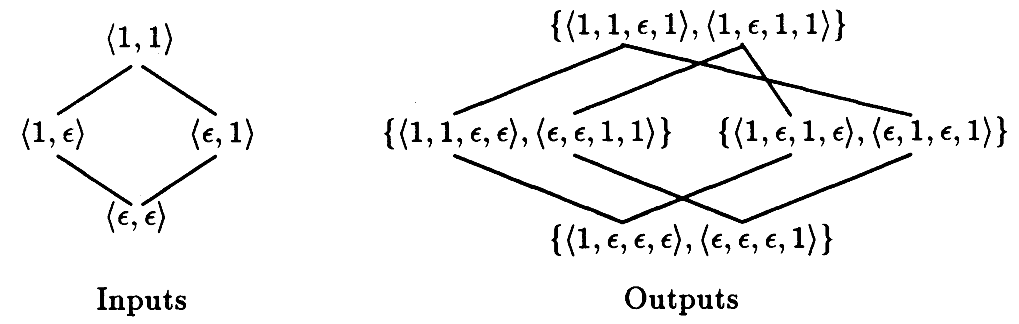
$\langle 1,1\rangle$
$\langle 1,\epsilon\rangle$
$\langle \epsilon,1\rangle$
$\langle \epsilon,\epsilon\rangle$
Inputs
$\{\langle 1,1,\epsilon,1\rangle,\langle 1,\epsilon,1,1\rangle\}$
$\{\langle 1,1,\epsilon,\epsilon\rangle,\langle \epsilon,\epsilon,1,1\rangle\}$
$\{\langle 1,\epsilon,1,\epsilon\rangle,\langle \epsilon,1,\epsilon,1\rangle\}$
$\{\langle 1,\epsilon,\epsilon,\epsilon\rangle,\langle \epsilon,\epsilon,\epsilon,1\rangle\}$
Outputs

Figure 6.1: An Egli-Milner monotone input-output relation from S^2 to S^4.

if and only if $i \sqsubseteq i'$ implies

$$\{o|(i,o) \in \mathcal{IO}[\![M]\!]\} \sqsubseteq_{EM} \{o'|(i',o') \in \mathcal{IO}[\![M]\!]\}.$$

Theorem 6.7. The class of oraclizable networks (and hence the infinity-fair merge networks) is a proper subclass of the class of networks with Egli-Milner monotone input-output relations.

Proof: It is easily seen that all oraclizable networks have Egli-Milner monotone input-output relations. To see that the containment is proper, consider the input-output relation described in Figure 6.1 (lines are drawn between comparable elements). It is clearly Egli-Milner monotone, but cannot be represented by any set of functions, since the "diamond" of relations among the inputs does not appear in the output. Since we know that sets of functions are universal for infinity-fair merge networks, no such network can implement this relation. ∎

Finally, we demonstrate the naturality of the relation by exhibiting a network built using angelic merge that computes it.

Example 6.1. Consider the network shown in Figure 6.2. The operation of

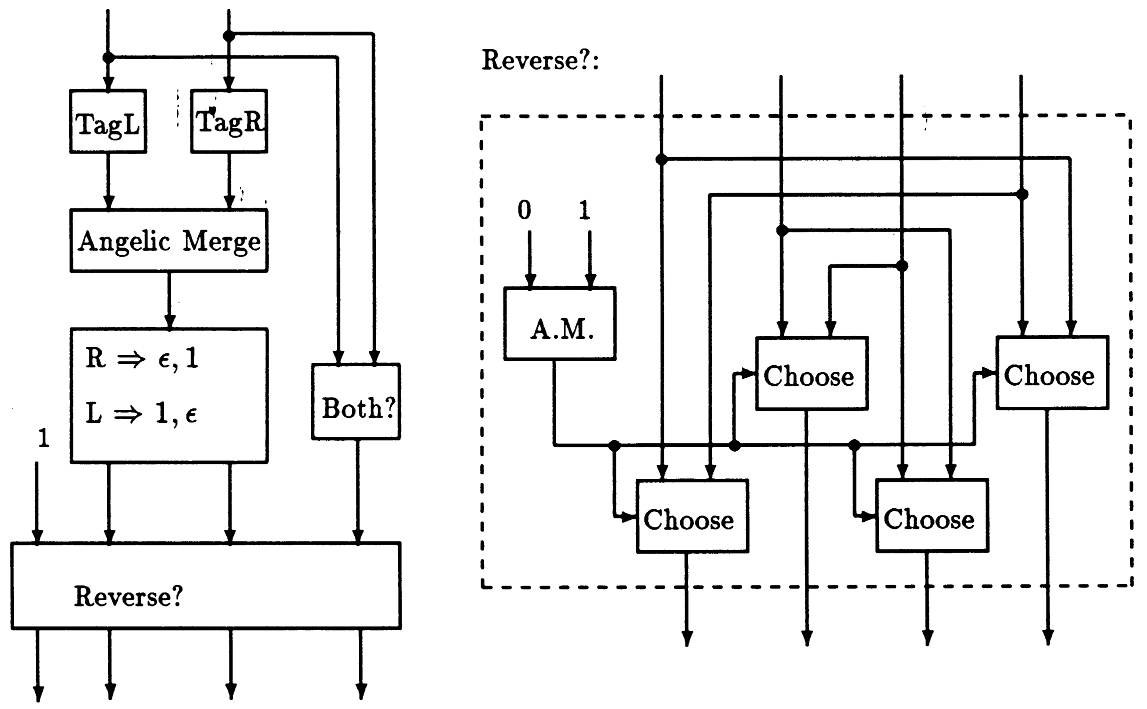

Figure 6.2: An angelic merge network that computes the given relation

each of the processes is as follows (the connections marked with •'s represent copy processes):

TagL, TagR: These tag the left and right inputs with an 'L' and 'R', respectively.

Angelic Merge: This is as described in Chapter 3. It will output whichever of its left or right channel that has data on it. If both channels are nonempty, they may be output in either order. The output of this merge is fed into a process that checks the tag on the first token it gets; if it is an 'L' it outputs a '1' on its left channel and nothing on its right, and if it is an 'R' it outputs a '1' on its right channel and nothing on its left.

Both?: This checks its inputs, and outputs a '1' only if it finds input on both input channels.

Reverse?: This randomly chooses whether to reverse the order of its four input channels. It is implemented as shown on the right of Figure 6.2. The angelic merge with inputs '0' and '1' will either produce '01' or '10', and this output is used by the four *Choose* processes. Each *Choose* process reads the first token of the input coming from the merge; if it is a '0' it will pass its left input, and if it is a '1' it will pass its right input.

It is easy to verify that the input-output relation of this network is that shown in Figure 6.1. ∎

The results of this chapter give us a clearer picture of the relationship of the oraclizable and Egli-Milner monotone networks with previously studied classes of indeterminate dataflow networks. This suggests a pictorial relation among the classes like that shown in Figure 6.3 (contrast this with Figure 3.4). In this figure each class contains those completely below it, and the classes defined by their implementation are more on the left, while classes defined by their input-output behavior are more on the right. The regions containing marks (●) are the ones that we know to be nonempty. This diagram illustrates what we know about the relationship between the specification of a network by its input-output behavior and the implementation of the networks, and it points out that there is still more to be discovered.

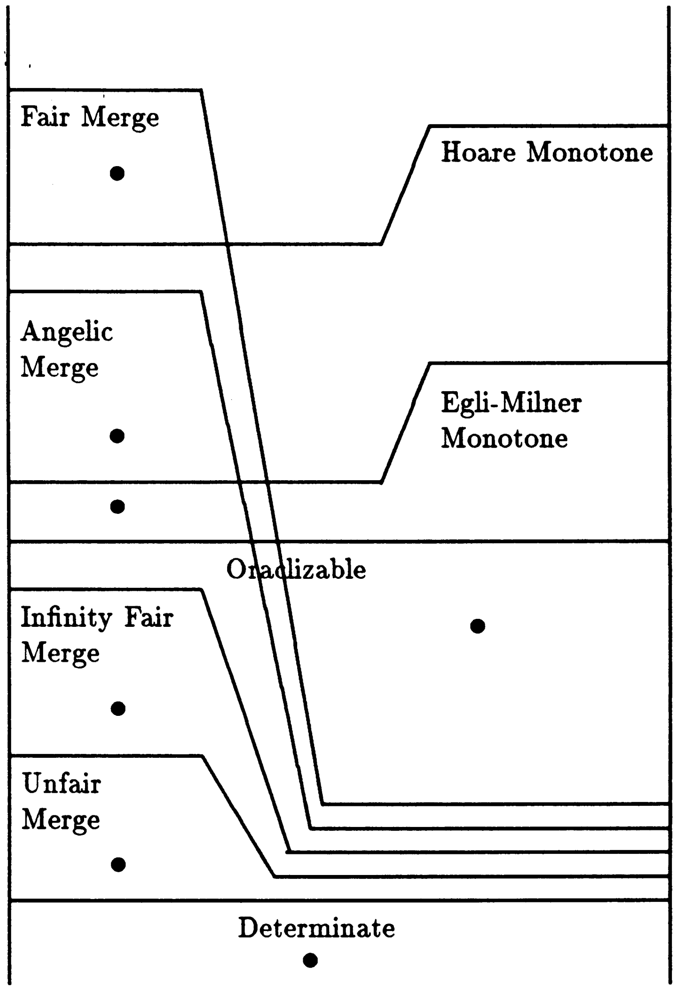

Figure 6.3: The inclusions among different classes of networks

Chapter 7

Related Work

In this chapter we present and discuss some of the work of other authors related to that presented in the last few chapters. In Chapter 2 we discussed work particularly relevant there, so in this chapter we will concentrate on related work in the dataflow setting.

Many semantic models for indeterminate dataflow networks have been developed [Pan85,BA81,KP85,Bro83,Pra86,Par82,SN85], but only recently have fully abstract models emerged [Jon89,JK88,Kok88,Rus89,RT88]. The full abstraction results of Chapter 4 are related to the work of Jonsson [Jon89], which we described in detail within the chapter. Panangaden and Shanbhogue [PS89] independently discovered the same full abstraction result as Jonsson, although using a slightly different formalism, and the results in Chapter 4 were also inspired by this work. Both of these papers used a distinguishing context based on one described by Kok [Kok88]. In his paper, Kok describes a semantic model for indeterminate dataflow networks based on tuples of infinite streams of finite strings of data tokens. This representation is closely related to traces, as is shown in [JK88], although it does not have the same linear interleaving structure. Kok's operational setting does not allow fair merge, but he uses the distinguishing context based on angelic merge to show that his semantics is fully abstract for those networks.

Rabinovich and Trakhtenbrot [RT88] have studied dataflow networks using a different observational philosophy than we do. In their work they consider a notion of observation in which only finite observations are allowed. They define a process by a prefix-closed set of finite strings (which are essentially what we call traces). This allows indeterminacy in their setting, but it means that they can only approximate infinite behaviors. As a result, for the purpose of distinguishing functional from nonfunctional behavior, the input-output relation is identified with the set of maximal (or limit) elements of the possible input-output behaviors. For example, a process with a single output channel that indeterminately chooses to output nothing or a single '1' is identified in their model with a process that will always output a '1'. Given this more restrictive notion of observation, the merge primitives introduced in Chapter 3 are not distinguishable in their model. The problem of finding a fully abstract semantics for this setting is both more and less difficult than for the setting we consider. In Rabinovich and Trakhtenbrot's setting with the restricted observations, it is harder to make observations to distinguish networks, but there are fewer observably different networks to distinguish among.

In [RT88], they describe an extended Kahn's principle for their setting. They discuss an adaptation of the Brock-Ackerman anomaly and the resultant modularity issues. They also show that their model of a process as a prefix-closed set of finite strings is fully abstract for their notion of observation. The method they employ, like that of Chapter 4, is based on verifying possible strings (traces) of a network. Given a possible string s for a network N, they define a network context to test that N behaves according to s, called an *s-tester*. The s-tester context connects to the input and output channels of N, and has only one external output. It proceeds by sending the tokens to the inputs of N specified in s, and verifying that the tokens on the output channels of N agree with s. As soon as all the data of s have been observed correctly, the s-tester emits a special token its external output channel. By design, the s-tester will emit the special token only if s is in their representation of N, and this suffices to distinguish any networks with different representations. This s-tester strategy is made possible by the fact that s is known to be finite.

With respect to related work that concentrates on fixed-point principles, Keller and Panangaden [KP86,Pan85] have developed models for the full range of indeterminacy that employ fixed-point constructions, but they are cumbersome and not fully abstract. Staples and Nguyen [SN85] describe a compositional model for indeterminate dataflow networks based on partially ordered multisets of input-output histories. This model has a fixed-point principle, but it is not fully abstract. Misra [Mis89] has described an equational system for reasoning about indeterminate networks in which network meanings are 'smooth' solutions to recursive equations, but he does not consider full abstraction, nor provide fixed-point techniques for computing the smooth solutions. Abramsky [Abr89] has developed a general categorical theory for Kahn-type models for indeterminate dataflow networks.

In several papers Broy [Bro83,Bro87,Bro88] has studied fixed-point semantics for indeterminate dataflow networks, but for the most part does not consider full abstraction. This work was done before the results of [Sta88,PS88,PS87] detailing the differences among the merge primitives, so his treatment of indeterminacy is somewhat inconsistent. In [Bro83] he describes a semantics for an applicative language for representing dataflow networks that contains McCarthy's ambiguity operator. The semantics is given via an intermediate description of the program constructs as sets of functional behaviors. The descriptions of most of the constructs are straightforward, but the description of the behaviors for the *amb* construct is dependent on the input to the program. We now know that such a dependence is unavoidable, since *amb* is capable of implementing angelic merge, but sets of functions cannot represent it. Broy's semantic representation is then given as the set of fixed points of the functional behaviors of the intermediate description. This representation is fully abstract for the language, but the intermediate description is not. A drawback of this representation is that the set of fixed points can only be constructed once and cannot be composed, and thus this representation fails to be compositional for complete networks.

In [Bro87] Broy describes semantic models for several classes of dataflow networks, including networks with unfair merge, infinite recursively defined networks, and networks with fair merge. One of the models he presents for the unfair merge networks is based on limit-closed sets of functions, and uses the corresponding fixed-point principle. To describe the fair merge networks he modifies this semantics as before so that the set of functions depends on the input to the network. He does this by including a predicate that limits the functions in the representation of fair merge to those that actually have fair behavior for the given input. He does not consider full abstraction for either of these models. In [Bro88] Broy provides an interesting discussion of different operational interpretations for the anomalies of Keller and Brock-Ackerman. However, he apparently was not aware that the Brock-Ackerman anomaly relies on the presence of fair or angelic merge, and the semantics he describes applies only to unfair merge.

Chapter 8

Conclusions

In this thesis we considered the problem of finding semantic models for indeterminacy that are both fully abstract and have fixed-point principles. We began by looking at a simple imperative language containing unbounded choice, and achieved a result that was as close to optimal as possible. We then moved on to the more general setting of dataflow networks and the hierarchy of indeterminate merge primitives. We showed that the straightforward generalization of Kahn's semantics based on the input-output relation fails to be compositional for any class of indeterminate dataflow networks. We then extended previous results and showed that a semantics based on traces is fully abstract for all indeterminate and determinate dataflow networks, thereby providing a model considerably more general than Kahn's.

This generalization has the drawback that it does not have a simple fixed-point principle. We then restricted our attention to the smaller class of oraclizable dataflow networks, and showed that for this class a generalization of Kahn's semantics to sets of functions is both fully abstract and has natural fixed-point principle. We then used this representation to compare the class of oraclizable networks to other classes, and discovered new relations among the classes.

A closely related direction for future work is to extend these results and develop semantic models for larger classes of indeterminate networks that are both fully abstract and have fixed-point principles. Possible approaches to this problem include introducing explicit timing information into the model, and employing category-theoretic constructions of fair merge as a limit.

The results of Chapter 6 demonstrated new relations between the specification of the input-output behavior of a network and its implementation in terms of the different primitives. This suggests another future direction of investigating further relations among the classes, including those defined by the weaker monotonicity properties and the possible addition of oracles to networks with fair or angelic merge. Also, it would be interesting to know under what conditions one can attain an exact correspondence between a extensional specification and an implementation class. Our definition of oraclizable put no restrictions on the set of streams that makes up the oracle. It is a separate and interesting study to explore classes in which the oracles are restricted to satisfy certain properties. Some specific work has been done in this area, as in [Sta89b] or [MPS88], but there is no doubt more to be discovered.

Another area of further study is the study of semantic models for other settings, and the relation of the results of this thesis to these models. Such settings may be extensions of the dataflow setting, such as one in which processes as well as data may be passed along channels, or one in which networks may be recursively defined. One may also consider extensions of the lambda calculus to include different kinds of concurrency and indeterminacy. There has been recent interest in concurrent constraint programming; this setting would also be interesting to study.

Bibliography

[Abr83] S. Abramsky. On semantic foundations for applicative multiprogramming. In J. Diaz, editor, *Proceedings of the Tenth International Conference On Automata, Languages And Programming*, pages 1–14, New York, 1983. Springer-Verlag.

[Abr89] S. Abramsky. A generalized Kahn principle for abstract asynchronous networks. In *Proceedings of the Fifth Conference on Mathematical Foundations of Programming Semantics*, 1989.

[AP86] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal Of The ACM*, 33(4):724–767, 1986.

[BA81] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts*, pages 252–259, 1981. LNCS 107.

[Bou80] G. Boudol. *Semantique Operationalle et Algebrique Des Programmes Recursifs Non-Deterministes.* Ph.D. dissertation, University de Paris VII, 1980. These d'Etat.

[Bro83] M. Broy. Fixed-point theory for communication and concurrency. In Bjoerner, editor, *Formal Description of Programming Concepts II*, pages 125–148. North-Holland, 1983.

[Bro87] M. Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[Bro88] M. Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[Coq88] T. Coquand. Categories of embeddings. In *Proceedings of the Third IEEE Symposium on Logic In Computer Science*, 1988.

[JK88] B. Jonsson and J. Kok. Comparing dataflow models. Manuscript, 1988.

[Jon89] B. Jonsson. A fully abstract trace model for dataflow networks. In *Proceedings of the Sixteenth Annual ACM Symposium On Principles Of Programming Languages*, 1989.

[Kah77] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[Kok88] J. Kok. Dataflow semantics. Technical Report CS-R8835, Centre for Mathematics and Computer Science, August 1988.

[KP85] R. M. Keller and P. Panangaden. Semantics of networks containing indeterminate operators. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 479–496, 1985. LNCS 197.

[KP86] R. M. Keller and P. Panangaden. Semantics of digital networks containing indeterminate operators. *Distributed Computing*, 1(4):235–245, 1986.

[Leh76] D. Lehmann. *Categories for Fixed-point Semantics*. Ph.D. dissertation, University of Warwick, 1976.

[LS89] N. A. Lynch and E. W. Stark. A proof of the Kahn principle for input/output automata. *Information and Computation*, 1989.

[LT87] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical Report MIT/LCS/TR-387, M. I. T. Laboratory for Computer Science, April 1987.

[Mil75] R. Milner. Processes: a mathematical model for computing agents. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium, Bristol, July 1973*, pages 157–173. North-Holland/American Elsevier, 1975.

[Mis89] J. Misra. Equational reasoning about nondeterministic processes. Manuscript, 1989.

[MPS88] D. McAllester, P. Panangaden, and V. Shanbhogue. Nonexpressibility of fairness and signaling. In *Proceedings of the 29th Annual Symposium of Foundations of Computer Science*, 1988.

[Pan85] P. Panangaden. Abstract interpretation and indeterminacy. In *Proceedings of the 1984 CMU Seminar on Concurrency*, pages 497–511, 1985. LNCS 197.

[Par82] D. Park. The "fairness problem" and non-deterministic computing networks. In *Proceedings of the Fourth Advanced Course on Theoretical Computer Science, Mathematisch Centrum*, pages 133–161, 1982.

[Plo76] G. D. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3):452–487, 1976.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Journal Of Parallel Programming*, 15(1):33–71, 1986.

[PS87] P. Panangaden and V. Shanbhogue. On the expressive power of indeterminate primitives. Technical Report 87-891, Cornell University, Computer Science Department, November 1987.

[PS88] P. Panangaden and E. W. Stark. Computations, residuals and the power of indeterminacy. In Timo Lepisto and Arto Salomaa, editors, *Proceedings of the Fifteenth ICALP*, pages 439–454. Springer-Verlag, 1988. Lecture Notes in Computer Science 317.

[PS89] P. Panangaden and V. Shanbhogue. Traces are fully abstract for networks with fair merge. Manuscript, 1989.

[RT88] A. Rabinovich and B. A. Trakhtenbrot. Nets of processes and dataflow. To appear in Proceedings of ReX School on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS, 1988.

[Rus89] J. R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of the 30th Annual Symposium of Foundations of Computer Science*, pages 170–177, 1989.

[Sha90] V. Shanbhogue. *The Expressiveness of Indeterminate Dataflow Primitives.* Ph.D. dissertation, Cornell University, 1990.

[Smy78] M. B. Smyth. Powerdomains. *Journal of Computer and System Sciences*, 16:23–36, 1978.

[SN85] John Staples and V. L. Nguyen. A fixpoint semantics for nondeterministic data flow. *Journal Of The ACM*, 32(2):411–444, April 1985.

[Sta87] E. W. Stark. Concurrent transition system semantics of process networks. In *Proceedings Of The Fourteenth Annual ACM Symposium On Principles Of Programming Languages*, pages 199–210, 1987.

[Sta88] E. W. Stark. On the relations computable by a class of concurrent automata. Technical Report TR 88-09, SUNY at Stony Brook, Computer Science Dept, 1988.

[Sta89a] E. W. Stark. Concurrent transition systems. *Theoretical Computer Science*, pages 221–269, 1989.

[Sta89b] E. W. Stark. A simple generalization of Kahn's principle to indeterminate dataflow networks. Technical Report TR 89-29, SUNY at Stony Brook, Computer Science Dept, Dec 1989.