

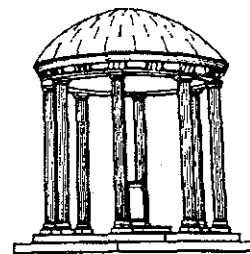
The Semantics of an FP  
Language with Infinite Objects

*TR88-022*

*April 1988*

*Teresa Anne Thomas*

The University of North Carolina at Chapel Hill  
Department of Computer Science  
CB#3175, Sitterson Hall  
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

The Semantics of an FP Language with Infinite Objects

by

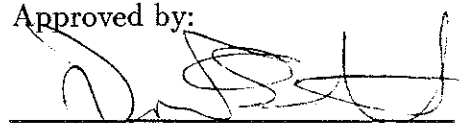
Teresa Anne Thomas

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

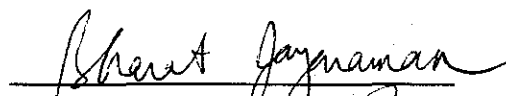
Chapel Hill

April 1988

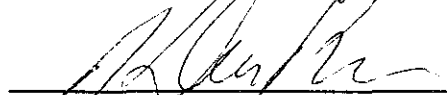
Approved by:



Advisor - Dr. Donald F. Stanat



Reader - Dr. Bharat Jayaraman



Reader - Dr. J. Dean Brock

©1988  
Teresa A. Thomas  
ALL RIGHTS RESERVED

TERESA ANNE THOMAS. The Semantics of an FP Language with Infinite Objects (Under the direction of DONALD F. STANAT.)

## Abstract

We describe an extension of Backus's FP (Functional Programming) languages to include infinite objects and discuss the semantic and mathematical issues surrounding the change. The extended languages, called SFP (Stream Functional Programming) languages, satisfy the following desiderata:

- The domain for FP is "embedded" in the new domain.
- The new domain contains infinite objects, both those infinite in length and those infinitely nested.
- The new language is not substantially different in syntax from Backus's language.
- The primitive functions of an FP language extend in an intuitively satisfying way to continuous functions on the new domain.
- The functional style of FP programming is preserved.
- The algebra of FP programs survives with few changes.

SFP differs from FP in that the domain for SFP contains infinite objects, approximations to complete objects, and  $\top$  (top), used to denote an error. Approximations to complete objects include  $\perp$  (bottom) and prefixes, where a prefix is a sequence that can be extended on the right. SFP uses a parallel outermost evaluation rule.

Any monotonic function on the FP domain can be extended to a continuous function on the SFP domain. We describe a domain of objects, a collection of functions, and two semantics, one denotational and one operational, for SFP. We show that the two semantics define nearly the same language. The definitions of the primitive functions and functional forms, according to both the operational and denotational semantics, are given, as well as example programs.

## **Acknowledgements**

Don Stanat suggested that I investigate adding infinite objects to FP, and it has been an interesting and fruitful topic. He has listened patiently to my questions and ideas, and the work has benefitted greatly from his comments and careful reading. Don has been quick to praise my efforts; I am grateful to him for his encouragement.

I would also like to thank the other members of my committee, Steve Weiss, Gyula Magó, and especially Bharat Jayaraman and Dean Brock, who probed, made suggestions, and answered my questions. Other faculty and staff have given their time to answer questions, and I am grateful for their assistance.

Finally, I am indebted to my fellow students. They have always been available when I needed help or encouragement.

## Table of Contents

Chapter 1 – Introduction .....	1
Motivation .....	1
A description of FP .....	2
Goal of this work .....	5
A characterization of the kind of solution being sought .....	7
Previous work .....	8
Novel aspects of this work .....	13
Summary of remaining chapters .....	13
Chapter 2 – The SFP Semantic Domains .....	15
Overview of the domains <b>O</b> , <b>B</b> , <b>C</b> , and <b>D</b> .....	16
Terminology and notation .....	18
The domains <b>O</b> and <b>B</b> .....	20
The domain <b>C</b> .....	21
( <b>C</b> , $\sqsubseteq$ ) is a complete lattice .....	25
The domain <b>D</b> .....	29
Summary .....	33
Chapter 3 – The Semantic Functions .....	34
Primitive functions on <b>D</b> .....	36
Functional forms .....	45
The collection of SFP functions .....	48
The meanings of SFP expressions .....	51
Continuity of SFP functions .....	52

Summary .....	55
Chapter 4 – Operational Semantics .....	56
The need for an operational semantics .....	56
The goal of an operational semantics .....	57
Definition of the operational semantics for SFP .....	58
Rewrite rules .....	59
Computation rule .....	62
Connection of the operational and denotational semantics .....	68
Summary .....	82
Chapter 5 – Algebra .....	83
Algebraic laws .....	84
Discussion .....	88
Conclusion .....	92
Chapter 6 – Examples .....	93
Chapter 7 – Conclusions and future work .....	101
Appendix A – FP .....	105
Appendix B – SFP – Denotational Semantics .....	110
Appendix C – SFP – Operational Semantics .....	118
Bibliography .....	127



# *Chapter 1*

## *Introduction*

### **Motivation**

Many computer scientists, including John Backus and David Turner [Backus 1978, Turner 1982], have argued that traditional programming languages are no longer adequate to meet the programming demands of today because they model too closely the operation of a von Neumann machine architecture. What is needed is a language which provides a powerful, tractable medium for fashioning algorithms. Functional programming languages are popular alternatives to traditional Pascal-like languages because they provide a better medium for reasoning about algorithms. Moreover, the advent of VLSI technology has freed architecture from the single-processor, separate large memory mold, and may make efficient implementation of non-von Neumann languages possible.

Backus's FP (Functional Programming) language described in his Turing Award Lecture [Backus 1978] has been a significant force in the increasing popularity of functional languages. The FP language enjoys a number of strengths. Its expressive power allows the programmer to manipulate a large number of objects with little code. Because it has sequences as a basic data type, the programmer is relieved of a lot of house-keeping details, such as setting and controlling loop limits. The language also uses powerful functional forms, or "program forming operations," which facilitate combining several small programs into a larger one. Because each FP program denotes a function in the mathematical sense, the collection of programs has nice mathematical properties. There is an algebra associated with the collection of programs that, in addition to facilitating proofs of correctness or equivalence, also makes possible automatic program transformation and optimization. Finally, FP

expresses parallelism in a natural way, and facilitates the investigation of equivalent but distinct forms of parallelism.

The FP language does have some deficiencies. The deficiencies of the language include the inability of programs to use the results of previous programs, no facilities for I/O, and the inability of programs to manipulate “infinite” data objects. Expanding the language to incorporate infinite objects is a first step towards making FP a viable language. This work describes an alteration of FP to include infinite objects and the semantic and mathematical issues surrounding such a change. The new extended language is called SFP for Stream Functional Programming. It retains the mathematical properties of FP.

## **A description of FP**

In order to understand the SFP language, it is necessary to have some knowledge of FP. Although there are many FP languages, we shall be concerned only with the FP language given by Backus in the Turing Award Lecture. Any reference herein to “FP” shall refer to precisely Backus’s language. However, the methods described here to extend Backus’s FP language can be used to extend any FP language.

A complete description of FP can be found in the Turing Award Lecture. A brief description is presented here.

There are many FP languages, but a particular one is specified by a particular choice of atoms, primitive functions and functional form operators. In this section we will describe the properties common to all FP languages. The examples we use to illustrate the properties will be drawn from the particular FP language defined by Backus.

An FP language is a language of expressions of two types. Object expressions denote objects, or data. Function expressions denote functions defined on the set of objects. An object expression is either

- a.  $\perp$ , (denoting an undefined result, etc.)

- b. an atom,
- c. a sequence  $\langle x_1, x_2, \dots, x_n \rangle$ , where each  $x_i$  is an FP expression, or
- d. an application  $f : x$ , where  $f$  is a function expression and  $x$  is an object expression.

An function expression is either

- a. a primitive function,
- b. a functional form, where a functional form consists of a functional form operator with function expression and object expression operands, or
- c. an identifier denoting a defined function, where a function definition is an expression of the form **Def**  $f = e$ , where  $f$  is a distinct identifier and  $e$  is a function expression.

An object is an object expression without any subexpressions that contain applications; thus an object does not contain any unevaluated subexpressions. Objects can be built inductively from  $\perp$ , the set of atoms, and the sequence constructor. The sequence constructor is  $\perp$ -preserving; that is, if any entry of a sequence is  $\perp$ , then the sequence is equal to  $\perp$ . In the sequel, we use the phrases “object expression” and “FP expression” interchangeably.

An FP program is a function expression. To apply a program  $f$  to input data  $x$ , we form the FP expression  $f : x$  and evaluate it.

In Backus’s FP language, the set of primitive functions includes functions for re-arranging data, performing arithmetic, and testing for various conditions. An example of a function that re-arranges data is *distl*, distribute from left, which when applied to the sequence  $\langle 5, \langle 2, 7, 9 \rangle \rangle$  results in the sequence  $\langle \langle 5, 2 \rangle, \langle 5, 7 \rangle, \langle 5, 9 \rangle \rangle$ . The usual arithmetic and boolean operations, such as addition, subtraction, *and*, *or*, *not*, etc., are included as primitive functions. Finally, some primitive functions allow testing for conditions, such as whether a sequence is empty (the primitive function *null*), whether two objects are equal (the primitive function

eq), and so on. More complicated conditions can be checked by combining these primitive functions by means of functional form operators.

Backus gave a number of functional form operators, which he calls *program forming operators*. For example, the functional form  $f \circ g$  is built from two functions,  $f$  and  $g$ , and the composition operator, and it denotes the function whose effect is the same as applying  $g$  to the argument, then applying  $f$  to that result. As another example, the functional form  $\bar{x}$ , where  $x$  is an object, denotes the function that returns  $\perp$  if the argument is  $\perp$  and the object  $x$  otherwise. The construction of a sequence of functions, denoted by  $[f_1, f_2, \dots, f_n]$ , applies each of its operands to a single argument and produces a sequence consisting of the results. The functional form *condition*, denoted  $p \rightarrow f;g$ , allows the programmer to choose what computation,  $f$  or  $g$ , will be performed on the argument based on the condition checked by the predicate,  $p$ . A complete list of primitive functions and functional form operators for FP can be found in Appendix A.

The semantics of an FP language can be given denotationally, as Backus did for the FFP language [Backus 1978], or operationally, as we do here. An FP expression is  $\perp$ , an atom, a sequence, or an application. The meaning of  $\perp$  is  $\perp$ ; the meaning of an atom is itself. The meaning of a sequence  $\langle x_1, x_2, \dots, x_n \rangle$  is  $\perp$  if the meaning of any  $x_i$  ( $1 \leq i \leq n$ ) is  $\perp$  and is the sequence of the meanings of its entries, otherwise. The meaning of an application  $f : x$  for any function  $f$  and any object  $x$  is found by replacing  $f : x$  with another expression, as described below, and finding the meaning of the new expression. The new expression used to replace  $f : x$  depends on  $f$  as follows:

1. If there is a definition **Def**  $f \equiv e$ , then  $f$  is replaced by  $e$ .
2. If  $f$  is a primitive function, then  $f : x$  is rewritten according to the definition of the primitive function.
3. If  $f$  is a functional form, then  $f : x$  is rewritten according to the definition of the functional form.

4. If none of the above classes apply, then the meaning of  $f : x$  is  $\perp$ .

If the process of rewriting an expression terminates, then the final expression is an object, which is the meaning of the expression. If the process of rewriting an expression does not terminate, then the meaning of the expression is defined to be  $\perp$ .

Thus,  $\perp$  is used to denote the meaning of a non-terminating computation. Backus also used  $\perp$  to denote an error, such as when the function  $+$  (denoting *addition*) is applied to an operand that is a sequence of characters. Rules to reduce such inappropriate applications to  $\perp$  are included in the definitions of the primitive functions and functional form operators.

### **Goal of this work**

This work has its goal the specification of an FP-like language that is applicable to both finite and infinite objects. A complete specification of a language requires defining the syntax and semantics of the language. BNF provides a uniform method for specifying the syntax of a language and is well understood. Specifying the semantics of a language is a much more difficult task, and there is not a standard method, such as using BNF to specify the grammar. When high-level languages first began to be developed, designers specified the meanings of the constructs of their languages informally, usually with a natural language. It became apparent that the use of a natural language to specify the meaning of a programming language is woefully unsatisfactory and inadequate. Thus language designers have increasingly turned to formal methods to specify the semantics of their languages.

Formal semantics come in a variety of flavors, and no one kind has been accepted as clearly superior to the others. The choice of which kind of formal method to employ is often dependent on the intended use of the semantics. Varieties of formal semantics include denotational, operational, axiomatic, and algebraic. In the specification of SFP, we have chosen to give two semantics, denotational and operational. The denotational semantics is mathematically precise and provides

a tractable medium for proving formal properties of the language and programs written in the language. It is easier to show, for example, that any program is continuous; proofs of correctness and equivalence of programs, where possible, will generally be easier. The operational semantics will show primarily how SFP could, in fact, be used as a basis for a machine implementation. It therefore gives much better guidance than the denotational semantics on implementation issues.

Often in mathematics a concept will have two (or more) characterizations that completely specify it, and since a concept can have only one definition, one of the characterizations is chosen for the definition and the other is shown to be equivalent to the definition. Similarly, the definition of SFP is its denotational semantics. The operational semantics is shown to be equivalent in some substantial sense to the denotational semantics. Thus the two semantics are reconciled by means of a consistency theorem.

The syntax chosen for SFP is a simple augmentation of that used for FP. Since syntax is easy to specify completely and unambiguously, it will receive little attention here. This work is devoted principally to the semantics of SFP.

The goal of this work is to show that FP can be extended to a new language, SFP, that has infinite objects and preserves the useful properties of FP. The work presented here satisfies the following desiderata:

- The domain for FP is “embedded” in the new domain.
- The new domain contains infinite objects, both those infinite in length and those infinitely nested.
- The new language is not substantially different in syntax from Backus’s language.
- The primitive functions of an FP language extend in an intuitively satisfying way to continuous functions on the new domain. For example,

$$(distl : \langle 4, \langle 1, 2, 3, \dots \rangle \rangle) = \langle \langle 4, 1 \rangle, \langle 4, 2 \rangle, \langle 4, 3 \rangle, \dots \rangle.$$

- The functional style of programming available with FP is preserved.
- The algebra of programs for FP survives with few changes.
- The language is suitable for use in an environment where an argument may be acquired over a long period of time, and the programs produce as much of the output as possible for any state of incomplete input.

This dissertation represents an extension of FP to a language that can be used to create programming environments. The work has focused heavily on the mathematical issues and given less attention to implementation details. Future work could include an implementation, such as incorporation of SFP into the FFP (Formal Functional Programming) machine [Magó 1979, 1980, Magó & Middleton 1984].

### **A characterization of the kind of solution being sought**

The design of a programming language involves both mathematical and computational concerns. These concerns sometimes pull a solution in opposite directions, such as when the mathematically elegant solution is impossible or impractical to implement, or when the efficient solution is mathematically untractable. But which concerns should be pre-eminent?

Historically, language designs have been compromised to accommodate practical concerns. The semantics of most languages have originally been given in English, and formal semantics have been imposed only after language definition. These formal semantics are quite cumbersome, partly because they are “add-ons” and largely because they are coupled to the notion of a machine state.

The design of SFP focuses on mathematical issues and restricts the influence of implementation issues to what is possible rather than the narrower notion of efficiency. SFP is not put forth as a language to be used in a real environment. Rather, SFP should be regarded as a first step.

## Previous work

The fundamental concepts used in the definition of SFP come from a number of sources. The FP language was described by Backus [Backus 1978]. The subject of denotational semantics was introduced by Scott [Scott 1971, 1972, 1976] and Strachey [Scott & Strachey 1971, Strachey 1966]. Stoy's text on the subject [Stoy 1977] gives a detailed treatment of the ideas of Scott and Strachey and shows how denotational semantics can be used as a tool for programming language definition. The ideas regarding fixed points for recursive equations, which are used to find solutions to those equations, are due to Tarski [Tarski 1955], who showed that such a solution always exists, and Kleene [Kleene 1952], who gave a constructive solution to these recursive equations.

Seminal work in streams was done by Burge [Burge 1975] and Kahn [Kahn 1974]. Burge defined functions, which he called streams, that allow efficient implementation of list processing yet are conceptually easy to program. It is often the case that one wishes to produce a list and then process each element of the list. If the list is completely produced in one phase and then processed later in a separate phase, then the intermediate result may require enormous storage. However, requiring the programmer to think at the same time about both producing the list and processing it increases the complexity of the problem with which he must deal. Burge's streams allow the programmer to program as if an intermediate list were produced, yet the implementation suppresses the creation of the intermediate list. The technique is essentially lazy evaluation, though Burge does not use that term.

Kahn [Kahn 1974] developed a simple language for parallel programming. His purpose was to illustrate a more formal, that is, mathematical, approach to designing languages. His system provides a facility for describing a parallel computation as a system of processors connected together by communication links, with an Algol-like program for each processor. Processors communicate with each other by means of FIFO queues, whose histories can be viewed as streams. His language controls the order of production and consumption of intermediate lists by means of two



functions, *send* and *wait*. Properties of the streams can be proved by using fixpoint equations, where the history of each of the communication lines is specified as one of the equations. One of the great strengths of his characterization of the behavior of the system is that it does not need to describe some immense, complex state. An example from this paper is used as the basis for an SFP example in Chapter 6.

More recently, a number of workers have addressed the problem of extending functional languages, including Backus's FP and FFP languages, to streams. Ida and Tanaka [Tanaka & Ida 1981 and Ida & Tanaka 1983] describe an extension of FP and FFP languages that includes streams. They are concerned primarily with programs they wish to write in their language, and so they offer a wide variety of programming examples. However, their description of the language is sketchy and informal, and they do not address such issues as whether their functions are continuous, how errors are handled, and the effect of the changes to the FP language on the algebra of programs.

Halpern *et al.* [Halpern *et al.* 1985] developed a semantics for a specific functional language with streams; their domain is similar to ours except that whereas we restrict extension of a sequence to its right end, their incomplete sequences are extensible at some single point which can be at the left end, the right end, or anywhere in the middle. The symbol  $\omega$  denotes the point in the sequence where extension can be done and is interpreted as a list of zero or more items yet to be computed. In order to simplify proofs, they do not allow  $\omega$  to occur in more than one place in the sequence. If  $\perp$  appears in a sequence, it represents exactly one item yet to be computed, and more than one  $\perp$  may appear in a sequence. Thus  $\perp$  represents an item about which nothing more than its existence is known.

Their work also uses  $\perp$  to represent "error." As a consequence, they are restricted operationally to withholding all output until enough of the argument has been revealed to ascertain that its structure is appropriate for the function being applied.

Their work treats functions as objects (“first-class citizens”), making it possible to create functions at run-time, whereas the SFP language presented here does not have that capability. Since functions can be higher-order, they make no distinction between primitive functions and functional forms.

The aim of Halpern *et al.* is to present a method of defining a semantics that is sound and complete. Their use of FP is incidental, since the method can be used in other language definitions. By soundness, they mean that any two expressions that are intended to be distinct are assigned different meanings. They show this by proving that the reduction rules preserve meaning, that is, if  $x \rightarrow y$  is a reduction rule, then  $\mu(x) = \mu(y)$  (“ $\mu$ ” is the meaning function). The proof consists of a case-by-case examination of each reduction rule. Completeness means that all expressions of the language can be reduced to their correct meanings. The proof is lengthy, involving a number of steps proving lemmas and propositions to get the main result, and it requires some case-by-case analysis of the reduction rules, though induction on the structure of the expressions is the main proof technique used.

The work of Halpern *et al.* in 1986 [Halpern *et al.* 1986] extends and modifies their earlier work [Halpern *et al.* 1985]. They give two new completeness criteria and suggest three rewrite strategies that correspond to the three types of completeness. As with the first paper, their set of primitive functions is sparse, including only **al** (append left), **ar** (append right), **first** (the FP selector 1), **last** (the FP selector  $1r$ ), **tl** (tail), **tr** (right tail), **null**, **cons** (the FP functional form Construction), **cond** (the FP functional form Condition), **comp** (the FP functional form Composition), **apply**, **K** (the FP functional form Constant), and **id**. Again, as with the first paper, their results apply only to the specific language they describe, although the results presumably could be extended. As before, they offer no example programs. The algebra of programs is not discussed in either paper.

Backus *et al.* [Backus *et al.* 1986, Williams & Wimmers 1988] describe a new functional language, FL (*F*unctional *L*evel), which is also based on FP. The aim of this work is to introduce I/O into FP in a restricted way. They contrast their

method with the unrestricted “convenient approach,” as used in ML [Milner 1984] and the “pure approach,” as used by Turner in SASL [Turner 1982]. The convenient approach is easy to use in programming, but it violates the property of functionality and hence complicates the underlying semantics. The pure approach maintains a simple semantics, but it requires more effort on the part of the programmer to manipulate the streams that are used to implement the I/O devices and history system. The authors argue that by restricting the way in which I/O is available to the program, they maintain a simple semantics while providing an easy to use system.

The FL language compromises these approaches. FL’s principal component is functional; actual computation is performed using only functional operations. But FL is not functional, since any computation also involves an underlying state which can be changed (only) by I/O operations. Their approach differs from the convenient approach of ML largely in that FL does not have an assignment operator; the designers of FL argue that inclusion of the assignment operator would greatly complicate the semantics of the language. Their approach differs from the pure approach in the the I/O operations are, in fact, not functional, as they are in SASL [Turner 1982].

The FL Language Manual [Backus *et al.* 1986] describes the FL language in detail. It contains twelve examples, none of which involve (useful) non-terminating computations. The focus of the subsequent paper [Williams & Wimmers 1988] is the question of how to properly handle I/O, particularly interactive I/O. They do not present any examples of non-terminating computations here, either, and to do so would have sidetracked them from their main objective. Apparently, streams can only be accessed implicitly through the history mechanism.

Dosch and Moller [Dosch & Moller 1984] present an extension of FP whose domain is based on term algebras. They exhibit a confluent and Noetherian term-rewriting system that is consistent and sufficiently complete with respect to the initial algebra.

Like ours, their domain includes all of the finite objects in FP, as well as infinite sequences and infinitely nested objects. These infinite objects are specified as solutions to recursive equations, whereas ours are specified as limits of chains of finite objects.

Like Halpern *et al.*, Dosch and Moller use bottom, denoted by  $\perp_{object}$  as the result when an error occurs. However, their work avoids the problem of having to withhold output until sufficient input has been acquired by using only a subset of the primitive functions; their examples involve only those primitive functions for which the problem does not arise.

Also used in their syntactic description of objects, as well as in the specification of complex functions, is an infix concatenation operator “&” that allows them to avoid the use of the prefix style *apndl*, which requires that the argument be a pair. Because & is infix, the problem of knowing that the argument of *apndl* or *apndr* is a pair does not occur; the syntax of & allows nothing else.

Explicit rules are given for *hd* (head, the same as the FP selector 1), *tl* (tail), *last* (the same as the FP selector 1r), and *tlr* (right tail). They briefly mention other FP primitive functions, and they state that rewrite rules could be given for them, but these other primitive functions are not used in their program examples.

They give a syntactic description of a *functional program*. A functional program consists of a finite list of definitions of the form **Def** *f*  $\equiv$  *e*, where *f* is an identifier and *e* is a functional form, and a single application *e*:*x*, where *e* is a functional form and *x* is an object.

Regarding the algebra of programs, they observe that laws in Backus’s algebra hold in their system; some can be strengthened from inequality in Backus’s system to equivalence in theirs.

Two evaluation schemes are discussed. A four step busy evaluation scheme and a five step lazy evaluation scheme are given informally, and they give an example of rewriting using each scheme. They also present a few trivial example programs.

Nickl [Nickl 1985] attempted to give a denotational semantics for the work of Dosch and Moller, but discovered that no consistently complete domain could be built for their term rewriting system. The difficulty apparently stems from the fact that sequences are extensible at both the right and left end. By requiring the left end of the sequence to be finite, that is, non-extensible, she produced a more restricted domain that is consistently complete.

### **Novel aspects of this work**

The present work differs from Burge and Kahn [Burge 1975, Kahn 1974] in that it extends the FP language, whereas the work of Burge and Kahn predates FP. The work of Ida and Tanaka [Tanaka & Ida 1981, Ida & Tanaka 1983] does not address any of the mathematical issues and does not attempt to give a formal semantics, as is done here.

The work of Halpern *et al.* [Halpern *et al.* 1985, 1986], Dosch and Moller [Dosch & Moller 1984], and Nickl [Nickl 1985] is much closer to this work than any other. This work differs from theirs in that it introduces a greatest element  $\top$  (top) to represent error. The use of  $\top$  makes it possible to produce better approximations to the output, given an approximation to the input, than can be done in the other systems. Another novel feature of the approach taken here is the use of a single mechanism to extend all primitive functions on the FP domain to primitive functions on the SFP domain. None of the other works offers a uniform mechanism; they must extend each primitive function individually. This mechanism also allows proofs of properties of the primitive functions to be handled generally, rather than on a case-by-case basis.

### **Summary of remaining chapters**

Chapter 2 gives the semantic domains of SFP. Chapter 3 describes the SFP functions and the denotational semantics of SFP expressions. Chapter 4 gives the operational semantics and shows how it is related to the denotational semantics. Chapter 5 discusses the changes to the algebra of programs. Chapter 6 presents

several program examples. Chapter 7 gives the conclusions and directions for future research. Throughout the document, definitions, lemmas, corollaries, and theorems are indexed in a common sequence; thus Definition  $X$  precedes Lemma  $X + K$  if  $K$  is a positive integer.

## Chapter 2

### The SFP Semantic Domains

An FP language is specified by a set of atoms, a set of primitive functions, a set of functional forms, and a meaning function. The goal in creating SFP is to change the definition of “object” so that infinite objects are also generated. This change will necessitate changes to the primitive functions, functional forms, and the meaning function. This chapter describes the changes to the object domain that results when infinite objects are added. The changes to the primitive functions, functional forms, and meaning function are the subject of Chapter 3.

Giving the denotational semantics of the SFP language requires defining the objects, the functions, and the meanings of all expressions of the language. For SFP, the set of objects will contain a least element  $\perp$ , a greatest element  $\top$ , atoms, finite sequences, infinite sequences, and approximations to sequences (called *prefixes*). The collection of functions will be specified by a set of primitive functions, a set of functional forms, and a mechanism for defining functions beyond the primitive ones. An expression in the SFP language is  $\perp$ ,  $\top$ , an atom, a prefix, a finite sequence, an infinite sequence, or an application, denoted  $f(x)$ , where  $f$  is a function expression (that is, a primitive function, a functional form, or an identifier), and  $x$  is an object expression (that is,  $\perp$ ,  $\top$ , an atom, a prefix, a finite sequence, or an infinite sequence). The meaning function  $\mu$  will specify the meaning of each SFP expression.

Along with the definition of the set of objects, we define a partial order  $\sqsubseteq$ , which provides the ability to compare the information content of various objects. Eventually, we will want to be able to distinguish between various states of the output of a computation as a function of time. For example, if a particular computation produces the natural numbers one at a time as a stream, that is, an infinite sequence,

we can denote the output after the first three have been produced as  $\langle 0, 1, 2 \rangle$  and later, assuming two more entries have been computed, as  $\langle 0, 1, 2, 3, 4 \rangle$ . We wish to express the fact that the output state  $\langle 0, 1, 2 \rangle$  contains less information than the output state  $\langle 0, 1, 2, 3, 4 \rangle$ , and the partial order gives us the means to do so by the expression  $\langle 0, 1, 2 \rangle \sqsubseteq \langle 0, 1, 2, 3, 4 \rangle$ .

Backus's domain is partially ordered by a trivial order under which each element except  $\perp$  is preceded only by itself and  $\perp$ , and  $\perp$  is preceded only by itself. (No other relationships exist.) The new SFP domains contain additional objects and the partial order is extended in a non-trivial way, as illustrated by the example of the preceding paragraph, to relate the new objects. The resulting structures are not *flat*, as Backus's domain is. (A flat domain is one in which the objects are unrelated to one another, except possibly to a least and/or a greatest element.) Furthermore, the new domains are complete lattices. (Backus's domain fails to be a complete lattice in that it lacks a greatest element.)

### Overview of the domains **O**, **B**, **C** and **D**

We wish to define the domain **D** for SFP so that it is a superset of Backus's domain, which he called **O**, and which we also call the FP domain. In order to give a uniform mechanism for translating Backus's primitive functions on his domain to primitive functions on **D**, we break the translation into three phases by creating two domains, **B** and **C**, which in complexity lie between the FP domain **O** and our target domain **D**. We have the following relationships:

$$\mathbf{O} \subset \mathbf{B} \subset \mathbf{C} \subset \mathbf{D},$$

where all containments are proper.

Conceptually we begin with a set of atoms **A** to construct a domain **O** whose objects include a least element **bottom**, denoted by  $\perp$ , the atoms, and the set of all finite sequences of objects from **O**.  $\perp$  is used in this domain to represent error



and the result of non-terminating computations. Sequences in  $\mathbf{O}$  are  $\perp$ -preserving, that is, any sequence containing  $\perp$  is equal to  $\perp$ .

We next construct the domain  $\mathbf{B}$ , which is similar to  $\mathbf{O}$  except that we add a greatest element, **top**, denoted by  $\top$ , which represents error. In the domain  $\mathbf{B}$ ,  $\perp$  no longer represents error as it does in  $\mathbf{O}$ . It is the least defined element; as such, it is used to denote a value about which we have no information, such as a non-terminating computation. Sequences in  $\mathbf{B}$  are  $\top$ -preserving, that is, if  $\top$  is an entry in the sequence, then the sequence is equal to  $\top$ . If a sequence does not contain  $\top$ , then it is  $\perp$ -preserving.

Each element in the domain  $\mathbf{O}$  is related by the partial order  $\leq$  only to itself and  $\perp$ . Each element in the domain  $\mathbf{B}$  is related by the partial order  $\sqsubseteq$  only to itself,  $\perp$  and  $\top$ . Both of these domains are flat, since all elements apart from  $\perp$  and  $\top$  are incomparable with each other.

We next construct an intermediate domain  $\mathbf{C}$  that contains, in addition to all the elements of  $\mathbf{B}$ , finite incomplete objects. These incomplete objects are approximations both to sequences, such as  $\langle 1, 2, 3 \rangle$ , and streams, such as  $\langle 1, 2, 3, \dots \rangle$ . As with  $\mathbf{B}$ ,  $\perp$  in  $\mathbf{C}$  is used to denote a value about which we have no information and  $\top$  is used to represent error. Sequences and incomplete objects in  $\mathbf{C}$  are  $\top$ -preserving, but in contrast to  $\mathbf{B}$ , they are not  $\perp$ -preserving. The partial order  $\sqsubseteq$  on  $\mathbf{B}$  is extended to  $\mathbf{C}$  so that incomplete objects precede the objects that they approximate.

The set of objects  $\mathbf{D}$ , which is the domain of SFP, contains finite and infinite objects and is constructed by forming the completion of the intermediate set  $\mathbf{C}$ , that is, by adding limit points to it. Thus, for each object in  $\mathbf{D}$ , the set  $\mathbf{C}$  contains approximations arbitrarily close to that object. The definition of the semantic domain  $\mathbf{D}$  is analogous to the definition of the real numbers by Cauchy sequences; a computational object in  $\mathbf{D}$  is defined as an equivalence class of infinite sequences of finite objects.

As a result of this definition of  $\mathbf{D}$ , sequences (both finite and infinite) and incomplete (both finite and infinite) objects are  $\top$ -preserving but not  $\perp$ -preserving. The partial order  $\sqsubseteq$  is extended so that limit points are preceded by all objects that approximate them.

The choice of  $\mathbf{C}$  as an extension of  $\mathbf{B}$  is novel and ensures the properties we seek; the completion of  $\mathbf{C}$  to get  $\mathbf{D}$  is a standard construction. Specifically,  $\mathbf{C}$  satisfies all our desiderata for an SFP domain except that it does not have infinite objects, and its completion,  $\mathbf{D}$ , satisfies all the desiderata. For each of the semantic domains  $\mathbf{B}$ ,  $\mathbf{C}$ , and  $\mathbf{D}$ , we define a partial order and show that it is a complete lattice.

### Terminology and notation

We first introduce some terminology and notation. An *atom* is a symbol used to represent an elementary datum; the set of atoms is nonempty and does not include the special elements  $\perp$  and  $\top$ .

For each of our domains, the term *object* denotes any element of the domain. These include  $\perp$ ,  $\top$ , atoms, and finite sequences for all the domains, as well as prefixes in  $\mathbf{C}$  and  $\mathbf{D}$ , and streams and infinitely nested sequences in  $\mathbf{D}$ . *Sequences* and *streams* are objects comprising a collection of objects (called *entries*), indexed by an initial segment of the natural numbers. A sequence has a finite number of entries, whereas the number of entries of a stream is equal to the cardinality of the integers. We denote sequences and streams with angle brackets; e.g.,

$$\langle 1, 2, 3 \rangle$$

and

$$\langle 1, 2, 3, \dots \rangle.$$

A *prefix* consists of a finite sequence of entries, but differs from a sequence in that it is an incomplete datum. Intuitively, a prefix can be closed to form a sequence

or it can be extended by adding another entry to its right end, giving a longer prefix. Prefixes serve the role of approximations to sequences or streams; in particular, each prefix is an approximation to a class of sequences and streams. When a prefix is extended by adding an additional element to its right end, the corresponding set of sequences and streams for which the resulting prefix is an approximation is reduced. Prefixes are denoted similarly to sequences, except that the right bracket is “}”. The prefix

$$\langle 1, 2, 3 \rangle$$

is an approximation to any sequence or stream whose first three entries are 1, 2, and 3, including the sequences  $\langle 1, 2, 3 \rangle$ ,  $\langle 1, 2, 3, 4 \rangle$ ,  $\langle 1, 2, 3, 4, 5 \rangle$  and  $\langle 1, 2, 3, 3, 3 \rangle$  and the streams  $\langle 1, 2, 3, 4, 5, 6 \dots \rangle$  and  $\langle 1, 2, 3, 3, 3, 3 \dots \rangle$ .

Prefixes are one kind of approximation. We are primarily interested in prefixes as approximations because they approximate streams. Other approximations are  $\perp$  and sequences that contain  $\perp$  or a prefix as a subexpression. For example, the sequence  $\langle \langle 1, 3 \rangle, \perp \rangle$  is an approximation to the sequence  $\langle \langle 1, 3 \rangle, \langle 2, 4 \rangle \rangle$ . Note that every incomplete object qualifies as an approximation.

Some of the objects are *complete*. Every atom is complete; a sequence (finite or infinite) is complete if each of its entries is complete;  $\top$  is complete; nothing else is complete. For example, the elements of  $\mathbf{D}$   $\langle \rangle$ ,  $\langle 1 \rangle$ ,  $\langle 1, \langle 2, \langle 3, \dots \rangle \rangle \rangle$ ,  $\langle 2, 4, 6, \dots \rangle$ , and  $\top$  are complete and  $\perp$ ,  $\langle \rangle$ ,  $\langle 1, \perp \rangle$ , and  $\langle \perp, \langle 1, \langle 1, \dots \rangle \rangle \rangle$  are not complete. Equivalently, an object is *incomplete* if it contains  $\perp$  or a prefix as a subexpression; otherwise, it is complete.

Finally, some notation employed here is new. Specifically,

$$\langle \langle \rangle_{i=m}^n \alpha_i = \langle \alpha_m, \dots, \alpha_n \rangle \text{ if } m \leq n,$$

$$\langle \rangle_{i=m}^n \alpha_i = \langle \alpha_m, \dots, \alpha_n \rangle \text{ if } m \leq n,$$

$$\langle \rangle_{i=m}^n \alpha_i = \langle \rangle \text{ if } m > n \text{ (the empty sequence), and}$$

$$\langle \rangle_{i=m}^n \alpha_i = \langle \rangle \text{ if } m > n \text{ (the empty prefix).}$$

## The domains $\mathbf{O}$ and $\mathbf{B}$

A set of atoms  $\mathbf{A}$  forms the basis for defining the domains  $\mathbf{O}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ , and  $\mathbf{D}$ . The set  $\mathbf{A}$  might be any non-empty set, but for concreteness we assume that it contains integers, characters, and the boolean values *true* and *false*.

The following is Backus's definition of the domain  $\mathbf{O}$  based on the set of atoms  $\mathbf{A}$ .

**Definition 0:** Let  $\mathbf{A}$  be a set of atoms. The domain  $\mathbf{O}$  based on  $\mathbf{A}$  is defined by the following:

- 0)  $\perp$  is in  $\mathbf{O}$ .
- 1) Every element of  $\mathbf{A}$  is in  $\mathbf{O}$ .
- 2) If  $0 \leq n < \infty$  and  $\alpha_i \in \mathbf{O}$  for  $1 \leq i \leq n$ , then  $\langle \alpha_i \rangle_{i=1}^n \in \mathbf{O}$ .

Any sequence containing  $\perp$  is equal to  $\perp$ .

- 3)  $\mathbf{O}$  has no elements other than those finitely constructible by the above.

The definition of  $\mathbf{B}$  adds the element  $\top$  to the domain of  $\mathbf{O}$ .

**Definition 1:** Let  $\mathbf{A}$  be a set of atoms. The domain  $\mathbf{B}$  based on  $\mathbf{A}$  is defined by the following:

- 0)  $\perp, \top$  are in  $\mathbf{B}$ .
- 1) Every element of  $\mathbf{A}$  is in  $\mathbf{B}$ .
- 2) If  $0 \leq n < \infty$  and  $\alpha_i \in \mathbf{B}$  for  $1 \leq i \leq n$ , then  $\langle \alpha_i \rangle_{i=1}^n \in \mathbf{B}$ . (We call these *sequences*.)

Any sequence containing  $\top$  is equal to  $\top$ , and, if it does not contain  $\top$ , a sequence containing  $\perp$  is equal to  $\perp$ .

- 3)  $\mathbf{B}$  has no elements other than those finitely constructible by the above.

Note that  $\mathbf{B} = \mathbf{O} \cup \{\top\}$ .

A *partial order* is a relation that is reflexive, anti-symmetric and transitive. If a set is partially ordered, and if every subset has a greatest lower bound (glb) and

least upper bound (lub) in the partially ordered set, then the partially ordered set is a *complete lattice*.

Backus gave a trivial partial order for  $\mathbf{O}$ . For every  $x \in \mathbf{O}$ ,  $\perp \leq x$  and  $x \leq x$ . That is, an element of  $\mathbf{O}$  is related only to itself and  $\perp$ .  $(\mathbf{O}, \leq)$  is not a complete lattice; most of its subsets do not have upper bounds.

The domain  $\mathbf{B}$  can be partially ordered by extending the order on  $\mathbf{O}$  so that every object lies below  $\top$ , giving a domain that is flat and is similar to  $\mathbf{O}$  except for the additional element  $\top$ . The addition of  $\top$  is sufficient to produce a complete lattice.

**Definition 2:** Let  $\mathbf{B}$  be based on a set of atoms  $\mathbf{A}$ . Then the relation  $\sqsubseteq$  is defined on  $\mathbf{B}$  as follows: For all  $\alpha, \beta \in \mathbf{B}$ ,  $\alpha \sqsubseteq \beta$  iff  $\alpha = \beta$ ,  $\alpha = \perp$ , or  $\beta = \top$ .

**Theorem 3:**  $\sqsubseteq$  is a partial order on  $\mathbf{B}$  and  $(\mathbf{B}, \sqsubseteq)$  is a complete lattice.

We omit the proofs, which are trivial, and proceed with the definition of  $\mathbf{C}$  and its partial order.

### The domain $\mathbf{C}$

The domain  $\mathbf{C}$  can be thought of as the set of all finite approximations, including all finite complete objects, to the elements of the domain  $\mathbf{D}$ . Definition 4 formalizes this concept.

**Definition 4:** Let  $\mathbf{A}$  be a set of atoms. The domain  $\mathbf{C}$  based on  $\mathbf{A}$  is defined by the following:

- 0)  $\perp, \top$  are in  $\mathbf{C}$ .
- 1) Every element of  $\mathbf{A}$  is in  $\mathbf{C}$ .
- 2) If  $0 \leq n < \infty$  and  $\alpha_i \in \mathbf{C}$  for  $1 \leq i \leq n$ , then
  - a)  $\langle \! \langle \alpha_i \rangle \! \rangle_{i=1}^n \in \mathbf{C}$ . (We call these *sequences*.)
  - b)  $\langle \! \langle \alpha_i \rangle \! \rangle_{i=1}^n \in \mathbf{C}$ . (We call these *prefixes*.)

Furthermore, any sequence or prefix containing  $\top$  is equal to  $\top$ .

(Note that sequences and prefixes are not  $\perp$ -preserving.)

3)  $\mathbf{C}$  has no elements other than those finitely constructible by the above.

Examples of elements from  $\mathbf{C}$  include  $\perp$ ,  $\mathbf{H}$ ,  $\langle\langle \vdash \rangle\rangle$ ,  $\langle \vdash \rangle$ ,  $\langle \perp \vdash \rangle$ ,  $\langle \mathbf{H}, \mathbf{M} \vdash \rangle$ , and  $\langle\langle \vdash \rangle, \langle \vdash \rangle\rangle$ . Note that if  $\mathbf{A}$  is a countable set, then  $\mathbf{O}$ ,  $\mathbf{B}$  and  $\mathbf{C}$  are countably infinite, but each element of these domains is itself finite.

Upon the set  $\mathbf{C}$  we impose a partial order relation,  $\sqsubseteq$ . As with the partial orders on  $\mathbf{O}$  and  $\mathbf{B}$ , the order can be thought of as “contains no more information than,” or “is an approximation to.” For the domain  $\mathbf{C}$  this order is very nearly the same as “is a prefix of;” this interpretation only fails for assertions involving  $\perp$  and  $\top$ .

**Definition 5:** Let  $\mathbf{C}$  be the set of objects based on  $\mathbf{A}$ . Then  $\sqsubseteq$  is an order defined on  $\mathbf{C}$  by the following:

- 0) For every  $\alpha \in \mathbf{C}$ ,  $\perp \sqsubseteq \alpha$  and  $\alpha \sqsubseteq \top$ .
- 1) For every  $\alpha \in \mathbf{A}$  and  $\beta \in \mathbf{C}$ ,  $\alpha \sqsubseteq \beta$  iff  $\alpha = \beta$  or  $\beta = \top$
- 2) If  $0 \leq n \leq m < \infty$ , and  $\alpha_i, \beta_i \in \mathbf{C}$  and  $\alpha_i \sqsubseteq \beta_i$  for  $1 \leq i \leq n$ , then
  - a)  $\langle\langle_{i=1}^n \alpha_i \rangle\rangle \sqsubseteq \langle\langle_{i=1}^m \beta_i \rangle\rangle$
  - b)  $\langle\langle_{i=1}^n \vdash \alpha_i \rangle\rangle \sqsubseteq \langle\langle_{i=1}^m \beta_i \rangle\rangle$
  - c)  $\langle\langle_{i=1}^n \vdash \alpha_i \rangle\rangle \sqsubseteq \langle\langle_{i=1}^m \vdash \beta_i \rangle\rangle$
- 3) For all  $\alpha, \beta \in \mathbf{C}$ ,  $\alpha \sqsubseteq \beta$  only if it is implied by the above.

From Definition 5 we can conclude  $\langle 1 \langle 2 \vdash \rangle \rangle \sqsubseteq \langle 1 \langle 2 \ 3 \vdash \rangle \rangle$ ,  $\langle \perp X \vdash \rangle \sqsubseteq \langle \perp X \rangle$ , and  $\langle 1 \ 2 \vdash \rangle \sqsubseteq \langle 1 \ 2 \ 3 \vdash \rangle$ . If  $\alpha \sqsubseteq \beta$ , we will say  $\alpha$  precedes  $\beta$ .

We present without proof several propositions that describe some of the properties of the order  $\sqsubseteq$  on  $\mathbf{C}$ .

**Proposition 6:** Nothing precedes  $\perp$  except itself.  $\top$  precedes nothing except itself.

**Proposition 7:** An atom is related only to itself,  $\perp$ , and  $\top$ .

**Proposition 8:** If  $\alpha$  is a prefix and  $\alpha \sqsubseteq \beta$ , then  $\beta$  is a prefix, a sequence, or  $\top$ .

**Proposition 9:** If  $\beta$  is a prefix and  $\alpha \sqsubseteq \beta$ , then  $\alpha$  is a prefix or  $\perp$ .

**Proposition 10:** An element  $\alpha$  can be refined to  $\beta$  if  $\beta \neq \top$ ,  $\alpha \sqsubseteq \beta$ , and  $\alpha$  is distinct from  $\beta$ . In this case,  $\alpha$  can be refined (transformed) to  $\beta$  by applying a sequence of *elementary refinements*, where an elementary refinement is one of the following:

- 0) An occurrence of  $\perp$  in  $\alpha$  is replaced by an atom or  $\langle \vdash$ .
- 1) An occurrence of  $\vdash$  in  $\alpha$  is replaced by  $\rangle$ .
- 2) An occurrence of  $\vdash$  in  $\alpha$  is replaced by  $\perp \vdash$ .

Note that no elementary refinement can be expressed as a sequence of other elementary refinements.

**Proposition 11:** If  $\beta$  is obtained from  $\alpha$  by a single elementary refinement, then  $\alpha \sqsubseteq \beta$  and for all  $\gamma$ , if  $\alpha \sqsubseteq \gamma$  and  $\gamma \sqsubseteq \beta$ , then either  $\alpha = \gamma$  or  $\gamma = \beta$ .

We next define what we mean by the *length* of an element of  $\mathbf{C}$  and by its *nesting level*. In the following definitions, let  $0 \leq n < \infty$  and  $\alpha_i \in \mathbf{C}$  for  $1 \leq i \leq n$ .

**Definition 12:** The *length* of an element of  $\mathbf{C}$  is defined as follows:

- 0) The length of  $\perp$  and all atoms is 0.
- 1) The length of  $\langle \underset{i=1}{\overset{n}{\alpha_i}}$  and  $\langle \underset{i=1}{\overset{n}{\vdash \alpha_i}}$  is  $n$ .

The length of  $\top$  is undefined.

**Definition 13:** The *nesting level* of an element of  $\mathbf{C}$  is defined as follows:

- 0) The nesting level of  $\perp$  and atoms is 0.
- 1) The nesting level of  $\langle \rangle$  and  $\langle \vdash$  is 1.
- 2) For  $n > 0$ , the nesting level of  $\langle \underset{i=1}{\overset{n}{\alpha_i}}$  and  $\langle \underset{i=1}{\overset{n}{\vdash \alpha_i}}$  is  $1 + \max_i \{\text{nesting level of } \alpha_i\}$ .

The nesting level of  $\top$  is undefined.

**Proposition 14:** If  $\alpha$  is a sequence of length  $n$  and  $\alpha \sqsubseteq \beta$ , then  $\beta$  is  $\top$  or a sequence of length  $n$ .

**Proposition 15:** If  $\alpha$  and  $\beta$  are sequences or prefixes and  $\alpha \sqsubseteq \beta$ , then the length of  $\alpha$  is less than or equal to the length of  $\beta$ .

**Proposition 16:** If  $\alpha$  and  $\beta$  are not  $\top$  and  $\alpha \sqsubseteq \beta$ , then the nesting level of  $\alpha$  is less than or equal to the nesting level of  $\beta$ .

**Theorem 17:**  $\sqsubseteq$  is a partial order on  $\mathbf{C}$ .

**Proof:** We must show that  $\sqsubseteq$  is reflexive, anti-symmetric, and transitive.

Let  $\alpha_1, \dots, \alpha_n, \beta_1, \dots, \beta_m, \gamma_1, \dots, \gamma_p, \alpha, \beta$ , and  $\gamma$  be elements of  $\mathbf{C}$ .

**Reflexive:** (For all  $\alpha \in \mathbf{C}$ ,  $\alpha \sqsubseteq \alpha$ .)

Case 1:  $\alpha = \perp$  or  $\top$ . Then  $\alpha \sqsubseteq \alpha$  by part 0 of Definition 5.

Case 2:  $\alpha$  is an atom. Then  $\alpha \sqsubseteq \alpha$  by part 1.

Case 3:  $\alpha = \langle \rangle$  or  $\langle \rangle$ . Then  $\alpha \sqsubseteq \alpha$  by parts 2a or 2c of Definition 5.

Case 4:  $\alpha = \langle \rangle_{i=1}^n \alpha_i$  or  $\langle \rangle_{i=1}^n \alpha_i$ . We will use induction on the nesting level of  $\alpha$  to establish that  $\alpha_i \sqsubseteq \alpha_i$  for  $1 \leq i \leq n$ , and hence,  $\alpha \sqsubseteq \alpha$ . Suppose that the nesting level of  $\alpha$  is 1. Then  $\alpha_1$  through  $\alpha_n$  are each  $\perp$  or an atom, and by Cases 1 & 3 above we have that  $\alpha_i \sqsubseteq \alpha_i$  for each  $i$ . Hence  $\alpha \sqsubseteq \alpha$  by parts 2a or 2c of Definition 5. For the induction hypothesis, assume that all sequences or prefixes with nesting level less than or equal to  $k$  precede themselves. If  $\alpha$  has nesting level of  $k + 1$ , then for each  $i$ ,  $\alpha_i$  has nesting level less than or equal to  $k$ . By the induction hypothesis,  $\alpha_i \sqsubseteq \alpha_i$ . Hence,  $\alpha \sqsubseteq \alpha$  by 2a or 2c of Definition 5.

**Anti-symmetric:** ( $[\alpha \sqsubseteq \beta \text{ and } \beta \sqsubseteq \alpha] \rightarrow \alpha = \beta$ .)

Case 1:  $\alpha = \perp$ .  $\beta \sqsubseteq \perp$  implies  $\alpha = \beta$  by Proposition 6.

Case 2:  $\alpha = \top$ .  $\top \sqsubseteq \beta$  implies  $\alpha = \beta$  by Proposition 6.



Case 3:  $\alpha$  is an atom.  $\alpha = \beta$  by Propositions 6 and 7.

Case 4:  $\alpha = \langle \rangle_{i=1}^n \alpha_i$  or  $\langle \rangle_{i=1}^n \alpha_i$ . Then  $\beta = \langle \rangle_{i=1}^n \beta_i$  or  $\langle \rangle_{i=1}^n \beta_i$ , and  $\alpha_i \sqsubseteq \beta_i$  and  $\beta_i \sqsubseteq \alpha_i$ . By induction on the nesting levels of  $\alpha_i$  and  $\beta_i$  we have that  $\alpha_i = \beta_i$ . Hence  $\alpha = \beta$ .

**Transitive:** ( $\alpha \sqsubseteq \beta$  and  $\beta \sqsubseteq \gamma \rightarrow \alpha \sqsubseteq \gamma$ .)

Case 1:  $\alpha = \perp$ . Then  $\alpha \sqsubseteq \gamma$  by part 0 of Definition 5.

Case 2:  $\alpha = \top$ .  $\alpha \sqsubseteq \beta$  implies  $\beta = \top$ , and  $\beta \sqsubseteq \gamma$  implies  $\gamma = \top$ . Therefore  $\alpha \sqsubseteq \gamma$ .

Case 3:  $\alpha$  is an atom. Then  $\alpha = \beta = \gamma$ , or  $\alpha = \beta \neq \gamma = \top$ , or  $\alpha \neq \beta = \gamma = \top$  by Proposition 7, and hence  $\alpha \sqsubseteq \gamma$ .

Case 4:  $\alpha = \langle \rangle_{i=1}^n \alpha_i$ . Then  $\beta = \langle \rangle_{i=1}^n \beta_i$  and  $\gamma = \langle \rangle_{i=1}^n \gamma_i$  such that  $\alpha_i \sqsubseteq \beta_i$  and  $\beta_i \sqsubseteq \gamma_i$ . By induction on the nesting level it can be shown that  $\alpha_i \sqsubseteq \gamma_i$  and hence,  $\alpha \sqsubseteq \gamma$ .

Case 5:  $\alpha = \langle \rangle_{i=1}^n \alpha_i$ . Then  $\beta$  and  $\gamma$  are prefixes or sequences such that  $\alpha_i \sqsubseteq \beta_i$  and  $\beta_i \sqsubseteq \gamma_i$ . In particular, the following combinations are possible, where  $n \leq m \leq p$ :

$$\beta = \langle \rangle_{i=1}^m \beta_i \text{ and } \gamma = \langle \rangle_{i=1}^p \gamma_i$$

$$\beta = \langle \rangle_{i=1}^n \beta_i \text{ and } \gamma = \langle \rangle_{i=1}^p \gamma_i$$

$$\beta = \langle \rangle_{i=1}^m \beta_i \text{ and } \gamma = \langle \rangle_{i=1}^m \gamma_i$$

For each of these cases, it can be shown by induction on the nesting level that  $\alpha_i \sqsubseteq \gamma_i$  and hence  $\alpha \sqsubseteq \gamma$ . ■

**(C,  $\sqsubseteq$ ) is a complete lattice**

We now establish that **C** is a complete lattice. It will suffice to show that **C** has a lub (least upper bound) and that every nonempty subset of **C** has a glb (greatest

lower bound) in  $\mathbf{C}$  [Jacobson 1974]. The next series of lemmas and corollaries provides the basis for proving that  $\mathbf{C}$  is a complete lattice.

**Lemma 18:** For every  $\alpha, \beta \in \mathbf{C}$ ,  $\alpha \sqcap \beta$  exists ( $\alpha \sqcap \beta$  denotes the glb of  $\alpha$  and  $\beta$ ).

**Proof:** By symmetry, we will only consider the cases on  $\alpha$ , and we will treat the cases sequentially, assuming that no previous case applies.

Case 1:  $\alpha = \perp$ . Then  $\alpha \sqcap \beta = \perp$ .

Case 2:  $\alpha = \top$ . Then  $\alpha \sqcap \beta = \beta$ .

Case 3:  $\alpha = \beta$ . Then  $\alpha \sqcap \beta = \alpha$ .

Case 4:  $\alpha$  is an atom. Then  $\alpha \sqcap \beta = \perp$  by Proposition 7.

Case 5:  $\alpha, \beta$  are prefixes or sequences. Only the following combinations will be considered. The rest follow from symmetry.

$$\alpha = \langle \rangle_{i=1}^n \alpha_i \text{ and } \beta = \langle \rangle_{i=1}^n \beta_i$$

$$\alpha = \langle \rangle_{i=1}^n \alpha_i \text{ and } \beta = \langle \rangle_{i=1}^m \beta_i, n \neq m$$

$$\alpha = \langle \rangle_{i=1}^n \alpha_i \text{ and } \beta = \langle \rangle_{i=1}^m \beta_i$$

$$\alpha = \langle \rangle_{i=1}^n \alpha_i \text{ and } \beta = \langle \rangle_{i=1}^m \beta_i$$

For the first combination, (where  $\alpha$  and  $\beta$  are sequences of the same length), let  $\gamma = \langle \rangle_{i=1}^n \alpha_i \sqcap \beta_i$ . For the other three combinations, let  $p = \min\{n, m\}$ , and  $\gamma = \langle \rangle_{i=1}^p \alpha_i \sqcap \beta_i$ . (To be a lower bound,  $\gamma$  must be a prefix if either  $\alpha$  or  $\beta$  is a prefix, or if  $\alpha$  and  $\beta$  are sequences of different lengths. If they are sequences of the same length,  $\gamma$  must be a sequence to be the glb.) We will first show (by induction on the nesting level) that  $\gamma$  exists and then show that  $\gamma = \alpha \sqcap \beta$ . Suppose that the nesting level of the  $\alpha_i$ 's and  $\beta_i$ 's is bounded by zero. Then all of the  $\alpha_i$ 's and  $\beta_i$ 's are  $\perp$  or atoms, and we have shown above that  $\alpha_i \sqcap \beta_i$  exists. Hence  $\gamma$  exists. For the induction hypothesis, assume that for any sequences or prefixes  $\alpha$  and  $\beta$ , if the nesting levels of  $\alpha$  and  $\beta$  are bounded by  $k$ , then  $\alpha \sqcap \beta$  exists. Now suppose that  $\alpha$  and  $\beta$  are sequences or prefixes

and that their nesting levels are bounded by  $k+1$ . Then by the definition of nesting level, each entry  $\alpha_i, \beta_i$  (for  $\alpha_i, \beta_i$  prefixes or sequences) has nesting level bounded by  $k$ . Hence each  $\alpha_i \sqcap \beta_i$  exists by the induction hypothesis, and hence  $\gamma$  exists. Finally, to see that  $\gamma$  is the glb, observe that it is a lower bound, and suppose that  $\delta$  is any lower bound. If  $\delta = \perp$ , then  $\delta \sqsubseteq \gamma$ . Else,  $\delta$  is a prefix or sequence, and its  $i^{\text{th}}$  entry precedes  $\alpha_i \sqcap \beta_i$ . Hence,  $\delta \sqsubseteq \gamma$ . ■

**Corollary 19:** Every finite set  $S \subseteq \mathbf{C}$  has a glb.

**Proof:** The proof is by induction on the size of the finite set. ■

The next two lemmas exhibit properties of  $\mathbf{C}$  that are used to establish that every nonempty subset of  $\mathbf{C}$  has a glb.

**Lemma 20:** Any element of  $\mathbf{C}$  not equal to  $\top$  is preceded by only a finite number of elements of  $\mathbf{C}$ .

**Proof:** We will use induction on the nesting level to establish the proposition. From the definition of  $\sqsubseteq$ , it is clear that  $\perp$  is preceded only by itself, and atoms are preceded only by themselves and  $\perp$ . Thus the assertion holds for objects of nesting level 0.  $\langle \rangle$  and  $\langle \rangle$  are preceded only by themselves,  $\langle \rangle$  and  $\perp$ . An element with nesting level equal to 1 is of the form  $\langle x_1 \dots x_n \rangle$  or  $\langle x_1 \dots x_n \rangle$ , where each  $x_i$  is  $\perp$  or an atom. These elements are each preceded by only a finite number of things, the prefixes of these elements and themselves. Suppose that any element with nesting level  $\leq k$  is preceded by only a finite number of things. We must show that an element with nesting level  $k+1$  is preceded by only a finite number of things. An element with nesting level  $k+1$  is of the form  $\langle x_1 \dots x_n \rangle$  or  $\langle x_1 \dots x_n \rangle$  where each  $x_i$  has nesting level  $\leq k$ .  $\langle x_1 \dots x_n \rangle$  is preceded only by elements of the set  $\{\langle y_1 \dots y_m \rangle \mid 0 \leq m \leq n \text{ and } y_i \sqsubseteq x_i\} \cup \{\langle y_1 \dots y_n \rangle \mid y_i \sqsubseteq x_i\} \cup \{\perp\}$ . For each  $i$ , there are only a finite number of possibilities for  $y_i$ , since, by the induction hypothesis,  $x_i$  is preceded by only a finite number of elements. Thus the set is finite. Similarly,  $\langle x_1 \dots x_n \rangle$  is preceded only by the elements of  $\{\langle y_1 \dots y_m \rangle \mid 0 \leq m \leq n \text{ and } y_i \sqsubseteq x_i\} \cup \{\perp\}$ , a finite set. ■

**Lemma 21:** Let  $S$  be a non-empty subset of  $C$  not containing  $\top$ . Then there exists a finite subset  $T$  of  $S$  such that the set of lower bounds of  $S$  is equal to the set of lower bounds of  $T$ .

**Proof:** The proof is by construction of  $T$ .  $S$  is not empty, so choose any element of  $S$ , call it  $\alpha$ . By Lemma 20,  $\alpha$  is preceded by no more than a finite number of things; let these be  $\{\beta_1, \beta_2, \dots, \beta_n\}$ . Thus  $\beta_i \sqsubseteq \alpha$ . Then let  $T = \{\alpha, \gamma_1, \gamma_2, \dots, \gamma_m\}$  such that for each  $\beta_i$  that is not a lower bound of  $S$ , there exists a  $\gamma_j \in S$  such that  $\beta_i$  does not precede  $\gamma_j$ . (We have that  $0 \leq m \leq n$ .)

We now show that the set of lower bounds of  $T$ , call it  $L_T$ , is equal to the set of lower bounds of  $S$ ,  $L_S$ . Since  $T \subseteq S$ , we have that  $L_S \subseteq L_T$ .  $L_T \subseteq L_S$  is true if  $\sigma \notin L_S \Rightarrow \sigma \notin L_T$ . Suppose  $\sigma \notin L_S$ . If  $\sigma$  does not precede  $\alpha$ , then  $\sigma \notin L_T$ . Otherwise,  $\sigma \sqsubseteq \alpha$ . Since  $\sigma$  is not a lower bound of  $S$ , there exists  $\gamma_j \in T$  such that  $\sigma$  does not precede  $\gamma_j$ . Therefore,  $\sigma \notin L_T$ . ■

**Corollary 22:** Every non-empty subset  $S$  of  $C$  not containing  $\top$  has a glb.

**Proof:** By Lemma 21, there exists a finite subset  $T$  of  $S$  such that the set of lower bounds of  $T$  is equal to the set of lower bounds of  $S$ . By Corollary 19,  $T$  has a glb. The glb of  $T$  is therefore the glb of  $S$ , since  $S$  and  $T$  have the same lower bounds. ■

**Lemma 23:** Let  $S$  be a subset of  $C$  containing  $\top$ . Then the glb of  $S$  is equal to the glb of  $S \setminus \{\top\}$ .

**Proof:** If  $S = \{\top\}$ , then  $\top = \text{glb } \{\top\} = \text{glb } \{ \} = \text{glb } S \setminus \{\top\}$ . Otherwise,  $S \setminus \{\top\}$  is not empty and by Corollary 22 has a glb, call it  $\alpha$ . To show that  $\alpha$  is the glb of  $S$ , we must show that  $\alpha$  is a lower bound of  $S$  and that for any other lower bound  $\beta$  of  $S$ ,  $\beta \sqsubseteq \alpha$ .  $\alpha$  is a lower bound of  $S$ , since  $\alpha$  precedes every element of  $S \setminus \{\top\}$  and  $\alpha \sqsubseteq \top$ . Let  $\beta$  be a lower bound of  $S$ . Then  $\beta$  is a lower bound of  $S \setminus \{\top\}$ , and hence,  $\beta \sqsubseteq \alpha$ . ■

**Corollary 24:** Every non-empty subset  $S$  of  $C$  has a glb.

**Proof:** Immediate from Corollary 22 and Lemma 23. ■

**Theorem 25:**  $(C, \sqsubseteq)$  is a complete lattice.

**Proof:** Since we have shown that  $\sqsubseteq$  is a partial order, it suffices to show that  $\mathbf{C}$  has a lub and that every non-empty subset of  $\mathbf{C}$  has a glb.  $\top$  is clearly the lub of  $\mathbf{C}$ , and Corollary 24 established that every non-empty subset of  $\mathbf{C}$  has a glb. ■

## The domain $\mathbf{D}$

The next step in defining the set of objects of the SFP domain  $\mathbf{D}$  is construction of the set of all chains of elements of  $\mathbf{C}$ . The expression  $\{\alpha_i\}$  denotes a sequence  $\{\alpha_i | i \in \mathbf{N} \ \& \ \alpha_i \in \mathbf{C}\}$ . The set of chains  $\mathbf{D}'$  is defined formally as:

$$\mathbf{D}' = \{ \{\alpha_i\} \mid \alpha_j \sqsubseteq \alpha_{j+1} \text{ for every } j \}.$$

Each element of  $\mathbf{D}'$  is called a *chain* (of elements of  $\mathbf{C}$ ). A chain is a sequence of approximations, where each approximation is at least as good as the preceding one.

Some examples of elements of  $\mathbf{D}'$  are:

$$\begin{aligned} & \{\perp, \perp, \perp, \dots\}, \\ & \{ \langle 1 \rangle, \langle 1 \ 2 \rangle, \langle 1 \ 2 \ 3 \rangle, \dots \}, \text{ and} \\ & \{ \perp, \langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle, \dots \}. \end{aligned}$$

We associate with each chain the (unique) element that is approximated arbitrarily well by the infinite sequence of approximations. The set  $\mathbf{D}'$  contains all of the kinds of objects we want, but the representations are not unique. For example, consider the following family of elements of  $\mathbf{D}'$ :

$$\begin{aligned} & \{\perp, 1, 1, 1, \dots\}, \\ & \{\perp, \perp, 1, 1, 1, \dots\}, \\ & \{\perp, \perp, \perp, 1, 1, 1, \dots\}, \end{aligned}$$

where the ellipses in each case denote an unending sequence of 1's. We wish to consider each of these to be an infinite representation of the finite object  $\mathbf{1}$ . To eliminate the problem of non-unique representation, we define an equivalence relation that groups together elements of  $\mathbf{D}'$  to reflect the desired notion of equality.

**Definition 26:** Let  $\{\alpha_i\}$  and  $\{\beta_i\}$  be elements of  $\mathbf{D}'$ . Then  $\{\alpha_i\} \sim \{\beta_i\}$  if for each  $k \in \mathbf{N}$ , there exists  $n \in \mathbf{N}$  such that  $\alpha_k \sqsubseteq \beta_n$  and  $\beta_k \sqsubseteq \alpha_n$ .

**Theorem 27:**  $\sim$  is an equivalence relation.

**Proof:** Reflexive: If  $\{\alpha_i\} \in \mathbf{D}'$ , we must find  $n$  for  $k$  such that  $\alpha_k \sqsubseteq \alpha_n$ . Choose  $n = k$ . Then by reflexivity of  $\sqsubseteq$ ,  $\sim$  is reflexive.

Symmetric: This is an immediate consequence of the definition of  $\sim$ .

Transitive: Suppose  $\{\alpha_i\} \sim \{\beta_i\}$  and  $\{\beta_i\} \sim \{\gamma_i\}$ . We want to show that  $\{\alpha_i\} \sim \{\gamma_i\}$ . Given  $k \in \mathbf{N}$ , we must find  $n \in \mathbf{N}$  such that  $\alpha_k \sqsubseteq \gamma_n$  and  $\gamma_k \sqsubseteq \alpha_n$ . Since  $\{\alpha_i\} \sim \{\beta_i\}$  there exists  $p \in \mathbf{N}$  such that  $\alpha_k \sqsubseteq \beta_p$ . Also, since  $\{\beta_i\} \sim \{\gamma_i\}$  there exists  $q, r \in \mathbf{N}$  such that  $\beta_p \sqsubseteq \gamma_q$  and  $\gamma_k \sqsubseteq \beta_r$ . Again, because  $\{\alpha_i\} \sim \{\beta_i\}$  there exists  $s \in \mathbf{N}$  such  $\beta_r \sqsubseteq \alpha_s$ . Let  $n = \max\{q, s\}$ . Then  $\beta_p \sqsubseteq \gamma_q \Rightarrow \beta_p \sqsubseteq \gamma_n$  and  $\beta_r \sqsubseteq \alpha_s \Rightarrow \beta_r \sqsubseteq \alpha_n$ . Hence we have that  $\alpha_k \sqsubseteq \beta_p$ , and  $\beta_p \sqsubseteq \gamma_n$ , which implies that  $\alpha_k \sqsubseteq \gamma_n$  by transitivity of  $\sqsubseteq$ . Similarly,  $\gamma_k \sqsubseteq \beta_r$  and  $\beta_r \sqsubseteq \alpha_n$  implies that  $\gamma_k \sqsubseteq \alpha_n$ . Hence,  $\sim$  is transitive.  $\blacksquare$

The quotient set  $\mathbf{D} = \mathbf{D}'/\sim$  is the desired domain. The order  $\sqsubseteq$  defined on  $\mathbf{C}$  induces a relation  $<$  on  $\mathbf{D}'$  and a partial order  $\sqsubseteq$  on  $\mathbf{D}$ . The relation  $<$  is a quasi-order (i.e., reflexive and transitive), but it is not anti-symmetric. In fact, the equivalence relation  $\sim$  is the link between  $<$  on  $\mathbf{D}'$  and  $\sqsubseteq$  on  $\mathbf{D}$ , being the minimum equivalence relation that produces anti-symmetry.

**Definition 28:** Let  $\alpha_i$  and  $\beta_i$  be elements of  $\mathbf{D}'$ . Then  $\{\alpha_i\} < \{\beta_i\}$  if for each  $k \in \mathbf{N}$ , there exists  $n \in \mathbf{N}$  such that  $\alpha_k \sqsubseteq \beta_n$ .

From the definition of  $<$ , it is apparent that if  $\alpha_i$  and  $\beta_i$  are elements of the same equivalence class, then  $\alpha_i \sqsubseteq \beta_i$ . Also, note that  $<$  is transitive.

**Definition 29:** Let  $\Gamma$  and  $\Delta$  be elements of  $\mathbf{D}$ , that is, they are equivalence classes. Then  $\Gamma \sqsubseteq \Delta$  if for every pair of chains  $\{\gamma_i\} \in \Gamma$  and  $\{\delta_i\} \in \Delta$ ,  $\{\gamma_i\} < \{\delta_i\}$ .

The next result establishes that in order to show  $\Gamma \sqsubseteq \Delta$ , we need not compare all the chains of  $\Gamma$  with all those of  $\Delta$ ; it suffices to compare any pair of representatives from the equivalence classes.

**Theorem 30:** Let  $\Gamma, \Delta$  be elements of  $\mathbf{D}$ . Then  $\Gamma \sqsubseteq \Delta$  iff there exist  $\{\gamma_i\} \in \Gamma$ ,  $\{\delta_i\} \in \Delta$  such that  $\{\gamma_i\} < \{\delta_i\}$ .

**Proof:**  $\Gamma$  and  $\Delta$  are non-empty; choose any chains  $\{\gamma_i\} \in \Gamma$  and  $\{\delta_i\} \in \Delta$ . That  $\Gamma \sqsubseteq \Delta$  implies  $\{\gamma_i\} < \{\delta_i\}$  follows immediately from Definition 29.

Now suppose that  $\{\gamma_i\} \in \Gamma, \{\delta_i\} \in \Delta$ , and  $\{\gamma_i\} < \{\delta_i\}$ . We must show that for any  $\{\eta_i\} \in \Gamma, \{\theta_i\} \in \Delta$ , it is true that  $\{\eta_i\} < \{\theta_i\}$ .  $\{\gamma_i\}$  and  $\{\eta_i\}$  are both elements of the same equivalence class, so, by Definitions 26 and 28,  $\{\eta_i\} < \{\gamma_i\}$ . Similarly,  $\{\delta_i\} < \{\theta_i\}$ . Since  $\{\gamma_i\} < \{\delta_i\}$ , by transitivity,  $\{\eta_i\} < \{\theta_i\}$ . ■

**Theorem 31:** The relation  $\sqsubseteq$  on  $\mathbf{D}$  is a partial order.

**Proof:** The proof is straightforward. ■

**Theorem 32:** If  $A$  is a countable set, then  $(\mathbf{D}, \sqsubseteq)$  is a complete lattice.

**Proof:** It suffices to show that  $\mathbf{D}$  has a glb and that any subset of  $\mathbf{D}$  has a lub. Consider the equivalence class of the chain  $\{\alpha_i \mid \alpha_i = \perp\}$ . (In fact, this chain is the only element of its equivalence class.) Clearly this object precedes everything, and the glb must precede this object, so this is the glb.

To show that any subset of  $\mathbf{D}$  has a lub, let  $S$  be a subset of  $\mathbf{D}$ . Choose one chain from each object in  $S$ . From these chains, we will construct a chain  $V$  whose equivalence class will be the lub of  $S$ . To show that the equivalence class of  $V$  is the lub of  $S$ , by Theorem 30 it will suffice to show that for each chosen chain  $\{\alpha_i\}$ ,  $\{\alpha_i\} < V$ , and that if for any other chain  $W$  such that  $\{\alpha_i\} < W$  for all chosen chains  $\{\alpha_i\}$ , then  $V < W$ .

We will construct  $V$  by first constructing an infinite series of sets whose elements come from the chosen chains. Then we will form a chain of lub's of elements of these sets by diagonalization.

Let  $X_k$  be the set of  $k^{th}$  elements of all of the chosen chains in  $S$ . Formally,

$$X_k = \{\alpha_k \mid \text{the chain } \{\alpha_i\} \text{ is one of the chosen chains}\}.$$

Each  $X_k$  is a subset of  $\mathbf{C}$  and therefore is countable, since  $A$  is countable.

Enumerate the  $X_k$ 's:

$$X_1 = \{x_{11}, x_{12}, x_{13}, \dots\}$$

$$X_2 = \{x_{21}, x_{22}, x_{23}, \dots\}$$

⋮

Construct the following sequence (to be shown a chain)  $V$  as a candidate for the lub of the chosen chains:

$$V = \{v_1, v_2, v_3, \dots\}, \text{ where}$$

$$v_1 = x_{11}$$

$$v_2 = x_{12} \sqcup x_{21} \sqcup v_1$$

$$v_3 = x_{13} \sqcup x_{22} \sqcup x_{31} \sqcup v_2$$

⋮

$$v_n = x_{1n} \sqcup x_{2(n-1)} \sqcup \dots \sqcup x_{n1} \sqcup v_{n-1}$$

⋮

All of these lub's exist, since  $\mathbf{C}$  is a lattice.  $V$  is a chain, since each element of  $V$  is the lub of the previous element and other elements, implying that the previous element certainly is a predecessor.

To show that  $V$  is the lub of the chosen chains, we can show that it is an upper bound and that it precedes all other upper bounds.  $V$  is an upper bound for the chosen chains if for all  $\{\alpha_i\}$  where  $\{\alpha_i\}$  is a chosen chain,  $\{\alpha_i\} < V$ .  $\{\alpha_i\} < V$  if for each  $\alpha_k$ , there exists  $v_n$  such that  $\alpha_k \sqsubseteq v_n$ . Consider  $\alpha_k$ . Since  $\alpha_k \in X_k$ ,  $\alpha_k = x_{k(i)}$  for some  $i$ . Choose  $n = k + i - 1$ . Then  $v_{(k+i-1)} = x_{1(k+i-1)} \sqcup x_{2(k+i-2)} \sqcup \dots \sqcup x_{k(k+i-k)} \dots \sqcup x_{(k+i-1)1} \sqcup v_{k+i-2}$ . The  $k^{\text{th}}$  element forming  $v_{(k+i-1)}$  is recognized as  $\alpha_k$ . ( $x_{k(k+i-k)} = x_{k(i)} = \alpha_k$ .) Clearly, then,  $\alpha_k \sqsubseteq v_{(k+i-1)}$ . Hence,  $V$  is an upper bound.

Now suppose that  $W = \{w_i\}$  is also an upper bound for the chosen chains, i.e., for every chosen chain  $\{\alpha_i\}$ ,  $\{\alpha_i\} < W$ . We must show that  $V < W$ , which will be true if for each  $k \in \mathbf{N}$ , there exists  $n \in \mathbf{N}$  such that  $v_k \sqsubseteq w_n$ . Observe that  $v_k = x_{1k} \sqcup x_{2(k-1)} \sqcup \dots \sqcup x_{k1} \sqcup v_{k-1}$ . Each  $x_{ij}$  making up  $v_k$  comes from some chosen chain. Since  $W$  is an upper bound for the chosen chains, each



$x_{ij}(i, j \leq k)$  precedes some element  $w_m$  in  $W$ . Choose such a  $w_m$  for each  $x_{ij}$ . The resulting set of  $k$   $w_m$ 's is totally ordered, since it is a subset of the chain  $W$ , and also is finite. Thus it has a maximum element, call it  $w_n$ . Since  $w_n$  is the maximum of the  $w_m$ 's, each of the  $x_{ij}$ 's precedes it. Therefore,  $v_k \sqsubseteq w_n$ , since  $v_k$  is the lub of the  $x_{ij}$ 's. ■

Thus we have established that the set  $\mathbf{D}$  under  $\sqsubseteq$  is a complete lattice, and this lattice will serve as the domain of a collection of continuous functions.

The definition of  $\mathbf{D}$  as a set of equivalence classes of chains gives us the ability to distinguish between *finite* and *infinite* objects. If an element of  $\mathbf{D}$  contains a chain  $\{\alpha_i\}$  such that  $\alpha_j = \alpha_k$  for all  $j, k \in \mathbf{N}$ , then that element is finite. Otherwise, the element is infinite. There is an obvious isomorphism between the finite objects and the set  $\mathbf{C}$ . Hence, we shall refer to the set of finite objects as  $\mathbf{C}$ .

### Summary

The SFP semantic domains  $\mathbf{B}$ ,  $\mathbf{C}$ , and  $\mathbf{D}$  have been defined based on a set of atoms  $\mathbf{A}$ . These domains are supersets of Backus's domain  $\mathbf{O}$ :

$$\mathbf{A} \subset \mathbf{O} \subset \mathbf{B} \subset \mathbf{C} \subset \mathbf{D},$$

$\mathbf{O}$  contains the atoms of  $\mathbf{A}$ , finite (complete) sequences, and  $\perp$ .  $\mathbf{B}$  contains all of the objects of  $\mathbf{O}$  and  $\top$ .  $\mathbf{C}$  contains all of the objects of  $\mathbf{B}$  and finite approximations.  $\mathbf{D}$  contains all of the objects of  $\mathbf{C}$  and infinite objects.

The partial order of  $\mathbf{O}$  has been extended to each of the new domains. Under the extended partial order, prefixes precede sequences and everything precedes  $\top$ .

Finally, the partially ordered sets  $(\mathbf{B}, \sqsubseteq)$ ,  $(\mathbf{C}, \sqsubseteq)$ , and  $(\mathbf{D}, \sqsubseteq)$  have been shown to be complete lattices. That  $(\mathbf{D}, \sqsubseteq)$  is a complete lattice will be particularly important in proving continuity of functions in the next chapter.

## Chapter 3

### *The Semantic Functions*

This chapter describes a way to alter the FP functions, which are defined on  $\mathbf{O}$ , to operate on all of the elements of the domain  $\mathbf{D}$ . The functions on  $\mathbf{D}$  will not, in general, be extensions of the functions on  $\mathbf{O}$ , since it is possible for a function on  $\mathbf{O}$  to map an element of  $\mathbf{O}$  to  $\perp$ , whereas the corresponding function on  $\mathbf{D}$  (as well as  $\mathbf{B}$  and  $\mathbf{C}$ ) maps that element to  $\top$ . This difference results from the fact that  $\perp$  represents error in  $\mathbf{O}$ , whereas  $\top$  represents error in  $\mathbf{D}$  (as well as  $\mathbf{B}$  and  $\mathbf{C}$ ).

In order to describe the functions on  $\mathbf{D}$ , we first specify a set of primitive functions and functional forms. We will establish that these functions have the desired properties of monotonicity and continuity. Once the primitive functions and functional forms have been given, the set of functions can be defined by describing the ways that the primitive functions and functional forms can be combined. We show that all of the functions thus defined are monotonic and continuous. Given the set of functions and using the objects of  $\mathbf{D}$  as defined in the previous chapter, we can finally give the meaning of any SFP expression using denotational semantics. This will complete our denotational description of the language.

In Chapter 2 the partial order  $\sqsubseteq$  was defined to describe the relationship between two objects, one of which approximates the other. This relative information content of various objects should be preserved by function application. Suppose we have two objects  $x_0$  and  $x_1$  such that  $x_0 \sqsubseteq x_1$ , that is,  $x_0$  contains no more information

than  $x_1$ . For any computationally useful function  $f$ , it is reasonable to expect that  $f(x_0) \sqsubseteq f(x_1)$ . For example, in the case that  $x_0$  and  $x_1$  are both approximations to some input argument  $x$ , this ensures that  $f(x_1)$  will contain at least as much information about  $f(x)$  as does  $f(x_0)$ . This property, called *monotonicity*, is true of all SFP functions, as we will show.

We defined infinite SFP objects as the limits of infinite chains of SFP objects; therefore, it is also necessary that all SFP functions be *continuous*. Continuity is the property that the image under  $f$  of the limit of a chain of approximations is equal to the limit of the images under  $f$  of the approximations in the chain. Continuity of functions is necessary to ensure that finite outputs produced by a computation from finite approximations to the input approximate the result of applying the function to the entire (possibly infinite) input.

One final property of SFP primitive functions ensures that the approximate results are as informative as possible. We want the result of applying an SFP primitive function to an approximate object to be as defined as possible, constrained only by the requirement that the primitive function be monotonic. This is accomplished by defining *maximal monotonic extensions*. For each approximate object, the primitive function is defined on that approximate object by taking the greatest lower bound of the images under that primitive function of the complete objects that are preceded by the approximate object in question. This formula is chosen because we want the image of the approximate object to precede the image of each complete object that is approximated. Using the set of lower bounds ensures monotonicity, and choosing the greatest of those ensures maximality.

## Primitive functions on $\mathbf{D}$

In this section we exhibit two related mechanisms for producing useful continuous functions over  $\mathbf{D}$  from monotonic functions on  $\mathbf{B}$ . A standard mechanism is used to extend any monotonic function over  $\mathbf{C}$  to a continuous function over  $\mathbf{D}$ . We introduce a second mechanism to extend any monotonic function over  $\mathbf{B}$  to a monotonic function over  $\mathbf{C}$ , which can then be extended by the first mechanism to a continuous function over  $\mathbf{D}$ . In order to use the original FP primitive functions given by Backus as a basis for the primitive functions on  $\mathbf{D}$ , we also need a definition that describes how any monotonic function on  $\mathbf{O}$  can be altered to a monotonic function on  $\mathbf{B}$ .

The combination of the definition for altering functions on  $\mathbf{O}$  to functions on  $\mathbf{B}$  and the two mechanisms is important because it provides a uniform way of creating a continuous function on  $\mathbf{D}$  from any monotonic primitive function on Backus's domain  $\mathbf{O}$ . This makes it feasible to define initially the primitive functions on  $\mathbf{O}$  and use the mechanisms to produce a corresponding continuous primitive function on  $\mathbf{D}$ . One only need ensure that the primitive function on  $\mathbf{O}$  is monotonic. That is not difficult; if the primitive function applied to  $\perp$  results in  $\perp$  (that is, if the function is bottom-preserving), then the function on  $\mathbf{O}$  is monotonic.

Earlier in this chapter, informal explanations of the concepts of monotonicity and continuity illustrated the need for all SFP functions to have these two properties. Here we formalize their definitions.

In Definitions 33, 34, and 35, let  $\mathbf{X}$ ,  $\mathbf{Y}$  and  $\mathbf{Z}$  be partially ordered sets under the partial order  $\sqsubseteq$ .

**Definition 33:** A function  $f$  from  $\mathbf{Y}$  to  $\mathbf{Z}$  is *monotonic* if for  $\alpha, \beta$  elements of  $\mathbf{Y}$ ,  
 $\alpha \sqsubseteq \beta \rightarrow f(\alpha) \sqsubseteq f(\beta)$ .

**Definition 34:** A set  $\mathbf{X}$  is *directed* if every finite subset of  $\mathbf{X}$  has an upper bound in  $\mathbf{X}$ .

**Definition 35:** A function  $f$  from  $\mathbf{Y}$  to  $\mathbf{Z}$  is *continuous* if for any directed set  $\mathbf{X} \subseteq \mathbf{Y}$ ,  $f(\text{lub } \mathbf{X}) = \text{lub}\{f(x) \mid x \in \mathbf{X}\}$ .

Because we are particularly interested in the primitive functions of Backus, we first show how, for any primitive monotonic function on  $\mathbf{O}$ , to produce a corresponding function on  $\mathbf{B}$ . This is done simply by modifying those primitive functions to reflect that  $\top$  represents error in the domain  $\mathbf{B}$ , a role played by  $\perp$  in  $\mathbf{O}$ ; this modification results in a primitive function on  $\mathbf{B}$ .

**Definition 36:** Let  $p$  be a monotonic function over  $\mathbf{O}$  (as is every Backus primitive function). The function  $\hat{p}$  from  $\mathbf{B}$  to  $\mathbf{B}$  is defined as follows:

$$\hat{p}(x) := \top, \text{ if } x = \top \text{ or } [p(x) = \perp \text{ and } x \neq \perp]$$

$$p(x) \text{ otherwise.}$$

In effect,  $\hat{p}$  is the same function as  $p$  except that  $\hat{p}$  uses  $\top$  for error rather than  $\perp$ . Note that if  $p$  is monotonic on  $\mathbf{O}$  (as is each Backus primitive function), then the resulting function  $\hat{p}$  on the flat lattice  $\mathbf{B}$  is also monotonic.

We now define *extension* and *maximal monotonic extension*.

**Definition 37:** Let  $(\mathbf{X}, \sqsubseteq)$ ,  $(\mathbf{Y}, \sqsubseteq)$  be partially ordered sets such that  $\mathbf{X} \subseteq \mathbf{Y}$  and let  $f$  be a monotonic function on  $\mathbf{X}$  and  $g$  a function on  $\mathbf{Y}$ . Then  $g$  is an *extension* to  $\mathbf{Y}$  of  $f$  on  $\mathbf{X}$  if  $f(x) = g(x)$  for every  $x \in \mathbf{X}$ . Furthermore,  $g$  is the *maximal monotonic extension* of  $f$  if  $g$  is monotonic, and for every other monotonic function  $h$  that is an extension to  $\mathbf{Y}$  of  $f$ ,  $h(x) \sqsubseteq g(x)$  for every  $x \in \mathbf{Y}$ .

**Proposition 38:** Let  $\mathbf{X}$ ,  $\mathbf{Y}$ ,  $f$ , and  $g$  be as in Definition 37. If  $g$  is the maximal monotonic extension of  $f$ , then  $g$  is unique.

**Proof:** To show that the maximal monotonic extension is unique, we must show that for any other maximal monotonic extension  $h$ ,  $g = h$ . Let  $h$  be a maximal monotonic extension of  $f$ . Since both  $g$  and  $h$  are maximal monotonic extensions of  $f$ , we have that for every  $x \in \mathbf{Y}$ ,  $h(x) \sqsubseteq g(x)$  and  $g(x) \sqsubseteq h(x)$ . Therefore,  $g = h$ . ■

When it is clear which domains are involved, we shall refer to  $g$  as simply the extension of  $f$  or the maximal monotonic extension of  $f$ .

Note that in general  $\hat{p}$  will not be an extension to  $\mathbf{B}$  of  $p$  on  $\mathbf{O}$ , since it is possible for  $p(x) = \perp$  and  $\hat{p}(x) = \top$  for some values of  $x \in \mathbf{O}$ . However, it is nearly an extension, the equality  $p(x) = \hat{p}(x)$  failing only for those elements that result in an error when  $p$  is applied to them. As noted before, the need for this modification arises from the fact that  $\top$  represents error in  $\mathbf{B}$  and  $\perp$  represents error in  $\mathbf{O}$ . Though not a true extension, we shall refer to  $\hat{p}$  as a modified extension or m-extension of  $p$ .

We next show how to extend a monotonic function on  $\mathbf{B}$  to a monotonic function on  $\mathbf{C}$ .

**Definition 39:** Let  $f$  be a monotonic function from  $\mathbf{B}$  to  $\mathbf{B}$ . Then  $f'$  is defined from  $\mathbf{C}$  to  $\mathbf{C}$  as:

$$f'(x) := \text{glb } \{f(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\}.$$

**Theorem 40:** Let  $f$  be a monotonic function from  $\mathbf{B}$  to  $\mathbf{B}$ . Then  $f'$  (as defined above) is a monotonic extension to  $\mathbf{C}$  of  $f$ .

**Proof:** We first show that  $f'$  is an extension of  $f$  by showing that for  $x \in \mathbf{B}$ ,  $f(x) = f'(x)$ . If  $x = \perp$  or  $x = \top$ , then it is clear that  $f(x) = f'(x)$ . Otherwise, if  $x \in \mathbf{B}$ , then the only values of  $y \in \mathbf{B}$  satisfying  $x \sqsubseteq y$  are  $x$  and  $\top$ . Thus

$$f'(x) = \text{glb}\{f(x), f(\top)\}$$

and since  $f$  is monotonic,  $f(x) \sqsubseteq f(\top)$ , so

$$= \text{glb}\{f(x)\}$$

$$= f(x).$$

Therefore,  $f'$  is an extension of  $f$ .

To show that  $f'$  is monotonic, suppose that  $a, b \in \mathbf{C}$  such that  $a \sqsubseteq b$ . Then

$$\{f(y) \mid a \sqsubseteq y \ \& \ y \in \mathbf{B}\} \supseteq \{f(y) \mid b \sqsubseteq y \ \& \ y \in \mathbf{B}\}.$$

Therefore

$$\text{glb}\{f(y) \mid a \sqsubseteq y \ \& \ y \in \mathbf{B}\} \sqsubseteq \text{glb}\{f(y) \mid b \sqsubseteq y \ \& \ y \in \mathbf{B}\}.$$

Hence

$$f'(a) \sqsubseteq f'(b).$$

■

The primitive functions as defined on  $\mathbf{C}$  are not usually extensions of the FP primitive functions on  $\mathbf{O}$  because when an error occurs, the FP primitive function produces  $\perp$ , whereas the primitive function on  $\mathbf{C}$  produces  $\top$ . If we restrict the domain to only those arguments that do not produce an error when used as an argument to the primitive function, then the primitive functions on  $\mathbf{C}$  will be

extensions of the FP primitive functions on these restricted domains. Note that different primitive functions may have different restricted domains because whether an error occurs for a particular argument depends on the primitive function being applied. For any monotonic function  $p$  on  $\mathbf{B}$ , we denote this restricted domain by  $B_p$ . ( $B_p \subseteq \mathbf{O} \subset \mathbf{B}$ .) In other words,  $B_p$  is simply the set of objects in Backus's domain that do not produce an error when used as arguments to the FP primitive function  $p$ .  $B_p$  contains exactly those elements for which  $\hat{p}$  produces the same result as  $p$ . The excluded elements are those that cause an error to occur when  $p$  or  $\hat{p}$  is applied to them.

**Lemma 41:** For any monotonic FP primitive function  $p$ , let  $B_p$  be the subset of  $\mathbf{O}$  such that  $x \in B_p$  if  $x = \perp$  or  $p(x) \neq \perp$ . Then  $\hat{p}$  is an extension to  $\mathbf{B}$  of  $p$  on  $B_p$  and  $\hat{p}'$  is the maximal monotonic extension to  $\mathbf{C}$  of  $p$  on  $B_p$ .

**Proof:** We first show that  $\hat{p}$  is an extension of  $p$  by showing that for every  $x \in B_p$ ,  $p(x) = \hat{p}(x)$ . Let  $x \in B_p$ . According to the definition of  $B_p$ ,  $x = \perp$  or  $p(x) \neq \perp$ . If  $x = \perp$ , then  $\hat{p}(x) = p(x)$  by Definition 36. If  $p(x) \neq \perp$ , then  $\hat{p}(x) = p(x)$ , again by Definition 36. Thus  $\hat{p}$  is an extension to  $\mathbf{B}$  of  $p$  on  $B_p$ . We have left to show that  $\hat{p}'$  is the maximal monotonic extension to  $\mathbf{C}$  of  $p$  on  $B_p$ . We must show that  $\hat{p}'$  is monotonic, that  $\hat{p}'$  is an extension to  $\mathbf{C}$  of  $p$  on  $B_p$ , and that for any other such extension  $h$  of  $p$ ,  $h(x) \sqsubseteq \hat{p}'(x)$  for every  $x \in \mathbf{C}$ .

We have seen that  $\hat{p}$  is monotonic, since  $p$  is monotonic. Furthermore, by Theorem 40,  $\hat{p}'$  is monotonic. According to Theorem 40,  $\hat{p}'$  is an extension to  $\mathbf{C}$  of  $\hat{p}$  on  $\mathbf{B}$ , which implies that  $\hat{p}'(x) = \hat{p}(x)$  for every  $x \in \mathbf{B}$  and thus for every  $x \in B_p$ . (Recall that  $B_p \subseteq \mathbf{O} \subset \mathbf{B}$ .) Therefore,  $\hat{p}'(x) = p(x)$  for every  $x \in B_p$ .



We have only to show that  $\hat{p}'$  is maximal. Suppose a monotonic function  $h$  is an extension to  $\mathbf{C}$  of  $p$  on  $\mathbf{B}_p$ . We must show that if  $x \in \mathbf{C}$ , then  $h(x) \sqsubseteq \hat{p}'(x)$ .

For every  $y \in \mathbf{B}_p$  such that  $x \sqsubseteq y$ , we have that  $h(x) \sqsubseteq h(y)$  because  $h$  is monotonic. Also, since both  $h$  and  $\hat{p}'$  are extensions of  $p$ , we know that  $p(y) = h(y)$  and  $p(y) = \hat{p}'(y)$ . Also, for  $y \in \mathbf{B}_p$  we have that  $p(y) = \hat{p}(y)$  by Definition 36. So  $h(y) = \hat{p}(y)$  for every  $y \in \mathbf{B}_p$ . Thus we can substitute  $\hat{p}(y)$  for  $h(y)$  in the relation  $h(x) \sqsubseteq h(y)$  to get  $h(x) \sqsubseteq \hat{p}(y)$  for every  $y \in \mathbf{B}_p$  such that  $x \sqsubseteq y$ . Thus  $h(x)$  is a lower bound for the set  $\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}_p\}$ , which means that  $h(x)$  precedes the glb of that set.

Since  $\mathbf{B}_p \subseteq \mathbf{B}$ , we have that

$$\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} = \{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}_p\} \cup \{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B} \setminus \mathbf{B}_p\}.$$

Suppose  $y \in \mathbf{B} \setminus \mathbf{B}_p$ . Then  $y \neq \perp$  and  $p(y) = \perp$ . From Definition 36 we have that  $\hat{p}(y) = \top$ . Therefore,  $\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B} \setminus \mathbf{B}_p\} = \{\top\}$  or  $\{\}$ . In either case, we have that  $\text{glb}\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} = \text{glb}\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}_p\}$

From the discussion concerning  $h(x)$  we have that  $h(x) \sqsubseteq \text{glb}\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}_p\}$ . Thus,  $h(x) \sqsubseteq \text{glb}\{\hat{p}(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} = \hat{p}'(x)$ , the result that we needed. ■

We have shown that  $\hat{p}'$  is the maximal monotonic extension to  $\mathbf{C}$  of  $p$  on  $\mathbf{B}_p$ . The use of  $\top$  as the error object, in conjunction with maximal monotonic extensions, permits results in some cases that are more defined than those achieved by similar languages described by others. Dosch & Moller [Dosch & Moller 1984] and Halpern

*et al.* [Halpern *et al.* 1985, 1986] use  $\perp$  to represent error. In order to ensure that functions are monotonic, the result of applying one of their functions to an input is  $\perp$  unless enough is known about the input to ascertain that no error can occur by further refinement of the input. To be more specific, consider the result of *apndl* applied to  $\langle 2 \ \langle 45 \ 10 \rangle \rangle$ . Since  $\langle 2 \ \langle 45 \ 10 \rangle \rangle \sqsubseteq \langle 2 \ \langle 45 \ 10 \rangle \ 33 \rangle$ , if  $\text{apndl}(\langle 2 \ \langle 45 \ 10 \rangle \ 33 \rangle) = \perp$  (denoting an error) in their systems, they have no choice but to define  $\text{apndl}(\langle 2 \ \langle 45 \ 10 \rangle \rangle) = \perp$  (denoting “no information”).

By contrast, since errors in SFP are denoted by  $\top$ , the SFP m-extension of *apndl* applied to  $\langle 2 \ \langle 45 \ 10 \rangle \rangle$  is able to produce the more informative result  $\langle 2 \ 45 \ 10 \rangle$  without sacrificing monotonicity.

We now use the fact that a monotonic function applied to a chain produces a chain to show that any monotonic function  $f$  on  $\mathbf{C}$  can be mapped to a continuous function  $f^*$  on  $\mathbf{D}$ .

**Definition 42:** Let  $f$  be monotonic on  $\mathbf{C}$ . Then the function  $f^*$  from  $\mathbf{D}$  to  $\mathbf{D}$  is defined as follows: If  $[\{\alpha_i\}]$  is the equivalence class of  $\{\alpha_i\}$ , and  $[\{f(\alpha_i)\}]$  is the equivalence class of  $\{f(\alpha_i)\}$ , then  $f^*([\{\alpha_i\}]) = [\{f(\alpha_i)\}]$ .

**Theorem 43:** Let  $f$  be a monotonic function on  $\mathbf{C}$ . Then  $f^*$  is well-defined.

**Proof:** To show that  $f^*$  is well-defined, we must show that if  $\{\alpha_i\}$  and  $\{\beta_i\}$  are members of the same equivalence class, then  $f^*([\{\alpha_i\}]) = f^*([\{\beta_i\}])$ , i.e.,  $[\{f(\alpha_i)\}] = [\{f(\beta_i)\}]$ . This will be true if  $\{f(\alpha_i)\} \sim \{f(\beta_i)\}$ .

For each  $k \in \mathbb{N}$ , we must find  $n \in \mathbb{N}$  such that the  $k^{\text{th}}$  element of  $\{f(\alpha_i)\}, f(\alpha_k)$ , precedes the  $n^{\text{th}}$  element of  $\{f(\beta_i)\}, f(\beta_n)$  and vice versa. Since  $\{\alpha_i\} \sim \{\beta_i\}$ , for each  $k \in \mathbb{N}$  there is  $n \in \mathbb{N}$  such that  $\alpha_k \sqsubseteq \beta_n$  and  $\beta_k \sqsubseteq \alpha_n$ . Because  $f$  is monotonic, we have that  $f(\alpha_k) \sqsubseteq f(\beta_n)$  and  $f(\beta_k) \sqsubseteq f(\alpha_n)$ . ■

We want to ensure that all primitive functions are monotonic and continuous on  $\mathbf{D}$ . Continuity implies monotonicity, so it will suffice to show that each function is continuous. However, the next theorem establishes that if a function  $f$  on  $\mathbf{C}$  is monotonic, then the corresponding function  $f^*$  is continuous on  $\mathbf{D}$ . We have seen earlier that for the FP primitive functions on  $\mathbf{O}$ , the corresponding functions on  $\mathbf{C}$  are monotonic if they are monotonic on  $\mathbf{O}$ . Therefore, any monotonic function on  $\mathbf{O}$  can be extended to a continuous function on  $\mathbf{D}$ . The only way a function on  $\mathbf{O}$  can fail to be monotonic is if it maps some non-bottom element to  $\perp$  and  $\perp$  to something other than  $\perp$ . Clearly any function on  $\mathbf{O}$  that maps  $\perp$  to  $\perp$  is monotonic; every FP primitive function is defined on  $\mathbf{O}$  and maps  $\perp$  to  $\perp$ . Therefore, every FP primitive function can be extended by Definitions 36, 39, and 42 to a continuous function on  $\mathbf{D}$ .

**Theorem 44:** Let  $f$  be a monotonic function on  $\mathbf{C}$ . Then  $f^*$  is continuous on  $\mathbf{D}$ .

**Proof:**  $f^*$  is continuous on  $\mathbf{D}$  if for every directed set  $X \subseteq \mathbf{D}$ ,  $f^*(\text{lub } X) = \text{lub } f^*(X)$ . The proof consists of establishing that  $f^*(\text{lub } X) \sqsubseteq \text{lub } f^*(X)$  and that  $\text{lub } f^*(X) \sqsubseteq f^*(\text{lub } X)$ .

$X$  and  $f^*(X)$  are sets of equivalence classes of chains, and  $\text{lub } X$ ,  $f^*(\text{lub } X)$  and  $\text{lub } f^*(X)$  are equivalence classes of chains. In view of the results of Theorem 30, we shall treat every equivalence class of chains as though it were a single (arbitrarily chosen) chain. Thus we shall treat  $X$  and  $f^*(X)$  as though they were sets of chains and  $\text{lub } X$ ,  $f^*(\text{lub } X)$  and  $\text{lub } f^*(X)$  as though they were simply chains. In the new denotations, the proof consists of showing that  $f^*(\text{lub } X) < \text{lub } f^*(X)$  and that  $\text{lub } f^*(X) < f^*(\text{lub } X)$ .

It will be true that  $f^*(\text{lub } X) < \text{lub } f^*(X)$  if for each point  $p$  on the chain  $f^*(\text{lub } X)$  there is a point  $q$  on the chain  $\text{lub } f^*(X)$  such that  $p \sqsubseteq q$ . If  $p$  is a point on  $f^*(\text{lub } X)$ , then there exists  $z$  on  $\text{lub } X$  such that  $p = f(z)$ . Since  $z$  is a point on the least upper bound of  $X$ ,  $z = \bigsqcup_{i=1}^n z_i$ , where each  $z_i$  is a point on some chain in  $X$ . Let  $Z_i$  be the chain containing  $z_i$ . (In fact, there may be several containing  $z_i$ , but we choose the one which causes  $z_i$  to participate in the lub at this point.) Since  $X$  is directed, the set  $\{Z_i | 1 \leq i \leq n\}$  has an upper bound in  $X$ , call it  $\{a_i\}$ . Since  $\{a_i\}$  is an upper bound for  $\{Z_i | 1 \leq i \leq n\}$ , for each  $i$ , there exists  $j \in \mathbb{N}$  such that  $z_i \sqsubseteq a_j$ . Let  $m = \max\{j \in \mathbb{N} | z_i \sqsubseteq a_j\}$ . For every  $i, 1 \leq i \leq n, z_i \sqsubseteq a_m$ . Therefore,  $\bigsqcup_{i=1}^n z_i \sqsubseteq a_m$ . Since  $f$  is monotonic,  $f(\bigsqcup_{i=1}^n z_i) \sqsubseteq f(a_m)$ , which implies that  $f(z) \sqsubseteq f(a_m)$ , which implies that  $p \sqsubseteq f(a_m)$ .  $f(a_m)$  is a point on the chain  $f(\{a_m\})$ , which is a chain in  $f^*(X)$ . There is a  $q$  in the chain  $\text{lub } f^*(X)$  such that  $f(a_m) \sqsubseteq q$ . This is the desired point, since  $p \sqsubseteq f(a_m)$  and  $f(a_m) \sqsubseteq q$  imply that  $p \sqsubseteq q$ .

It will be true that  $\text{lub } f^*(X) < f(\text{lub } X)$  if for each point  $z$  on  $\text{lub } f^*(X)$  there is a point  $f(q)$  on  $f(\text{lub } X)$  such that  $z \sqsubseteq f(q)$ . Since  $z$  is a point on a least upper bound, it can be written as  $z = f(z_1) \sqcup f(z_2) \sqcup \dots \sqcup f(z_n)$ . Each  $f(z_i)$  is a point on some chain in  $f^*(X)$ , so each  $z_i$  is a point on some chain in  $X$ . Each  $z_i$  precedes some  $y_i$  where  $y_i$  is a point on  $\text{lub } X$ , since  $\text{lub } X$  is an upper bound for  $X$ . Choose the largest such  $y_i$ , call it  $q$ . Now  $z_i \sqsubseteq q$  for every  $i$ , which implies that  $\bigsqcup_{i=1}^n z_i \sqsubseteq q$ . Because  $f$  is monotonic, we have that  $f(\bigsqcup_{i=1}^n z_i) \sqsubseteq f(q)$ . Therefore,  $z \sqsubseteq f(q)$ , and we have shown that  $f(q)$  is the desired point. ■

We have now completed the description of how primitive functions over  $\mathbf{O}$  can be used to construct primitive functions over  $\mathbf{D}$ . Complete definitions of all of the SFP primitive functions are given in Appendix B.

Not all functions on  $\mathbf{C}$  can be extended by Definition 42 to functions on  $\mathbf{D}$ . If a function  $f$  is not monotonic on  $\mathbf{C}$ , the definition of  $f^*$  fails to produce a function on  $\mathbf{D}$  because the image of some chain under  $f$  fails to be a chain itself. Consider some such  $f$ . Since  $f$  is not monotonic, there exist  $x, y \in \mathbf{C}$  such that  $x \sqsubseteq y$  and  $f(x) \not\sqsubseteq f(y)$ . Therefore, the image of any chain containing  $x$  and  $y$  is not a chain. Each of the following functions on  $\mathbf{C}$  are not monotonic and therefore cannot be extended by Definition 42 to functions on  $\mathbf{D}$ :

a function that closes a prefix (converts a prefix into a sequence),

a function that measures the length of a prefix,

a function that tests whether an argument is a prefix,

a function that tests whether an argument is the null prefix.

We are only interested in monotonic functions, however, so the fact that not all functions are extendable by Definition 42 is of little consequence. Every monotonic function on  $\mathbf{C}$  can be extended by Definition 42 to a monotonic function on  $\mathbf{D}$ .

### Functional forms

In order to combine primitives into useful programs, we need a set of combining, or functional, forms. We restrict ourselves here to extensions of the functional forms found in the Turing Lecture [Backus 1978], plus one variation, to operate on functions on  $\mathbf{D}$ . The definitions of some of those forms, such as *composition* and *construction*, hold in  $\mathbf{D}$  without modification. The definitions of others, such as

*insert* and *apply-to-all*, depend on the structure of the argument, and thus need to be modified to define their action on the elements of  $\mathbf{D}$  that are not in Backus's domain (i.e., approximations,  $\top$ , and streams), but the new definitions are straightforward modifications.

The definition of the functional form *condition* does not depend on the structure of the argument, but the definition must be modified, anyway, primarily because "error" has changed from  $\perp$  to  $\top$ . The functional form *constant* must also be altered slightly for the same reason.

Finally, we include another *insert* functional form, sometimes referred to elsewhere as *left insert*.

The definitions given for *insert* and *apply-to-all* are for the functional forms on  $\mathbf{C}$ . Using these functional forms on elements of  $\mathbf{D}$  merely requires applying the functional expression to each element of each chain of the elements of  $\mathbf{D}$ , the standard method for extending functions on  $\mathbf{C}$  to functions on  $\mathbf{D}$ . The other definitions work for both the domains  $\mathbf{C}$  and  $\mathbf{D}$ .

### Composition

$$(f \circ g)(x) \equiv f(g(x))$$

### Construction

$$[f_1, \dots, f_n](x) \equiv \langle f_1(x), \dots, f_n(x) \rangle$$

### Condition

$$(p \rightarrow f; g)(x) \equiv p(x) = T \rightarrow f(x);$$

$$p(x) = F \rightarrow g(x);$$

$$p(x) = \perp \rightarrow \perp;$$

$\top$

### Constant

$$\bar{x}(y) \equiv y = \top \rightarrow \top;$$

$x$

### Insert

Right Insert (from the original FP Insert)

$$/f(x) \equiv x = \perp \rightarrow \perp;$$

$$x = \langle x_1 \rangle \rightarrow x_1;$$

$$x = \langle x_1, \dots, x_n \rangle \& n \geq 2 \rightarrow f(\langle x_1, /f(\langle x_2, \dots, x_n \rangle) \rangle);$$

$$x = \langle x_1, \dots, x_n \rangle \& n \geq 0 \rightarrow \perp;$$

$\top$

Left Insert

$$\backslash f(x) \equiv x = \perp \rightarrow \perp;$$

$$x = \langle x_1 \rangle \rightarrow x_1;$$

$$x = \langle x_1, \dots, x_n \rangle \& n \geq 2 \rightarrow f(\langle \backslash f(\langle x_1, \dots, x_{n-1} \rangle), x_n \rangle);$$

$$x = \langle x_1, \dots, x_n \rangle \& n \geq 0 \rightarrow \perp;$$

$\top$

Apply to all

$$\alpha f(x) \equiv x = \perp \rightarrow \langle \rangle;$$

$$x = \phi \rightarrow \phi;$$

$$x = \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

$\top$

Binary to unary

$$(bu f x)(y) \equiv f(\langle x, y \rangle)$$

**While**
$$(while\ p\ f)(x) \equiv p(x) = T \rightarrow (while\ p\ f)(f(x));$$
$$p(x) = F \rightarrow x;$$
$$\top$$
**The collection of SFP functions**

A *function* of the SFP language is denoted by a *function expression*, which is a primitive function, a functional form, or an identifier  $f$  denoting a defined function. A defined function is declared by a *defining equation* of the form **Def**  $f := exp$ , where  $f$  is a function variable name and  $exp$  is a function expression. The grammar below generates  $F$ , the set of all possible function expressions for SFP. We do not allow any function variable name to be used as the left-hand side of more than one defining equation.

PRIMITIVE ::= primitive function

OBJ ::= object

FUN-VAR ::=  $f \mid g \mid h \mid \dots$  (non-subscripted character strings not already used as atoms or primitive function names)

F-LIST ::=  $F \mid F, F\text{-LIST}$



$F ::= \text{PRIMITIVE} \mid$   
 $\text{FUN-VAR} \mid$   
 $(F \circ F) \mid$   
 $(F \rightarrow F; F) \mid$   
 $[ \text{F-LIST} ] \mid$   
 $(/F) \mid$   
 $(\backslash F) \mid$   
 $\overline{\text{OBJ}} \mid$   
 $(\alpha F) \mid$   
 $\text{while}(F F) \mid$   
 $\text{bu}(F \text{ OBJ})$

An environment is a collection of defining equations and the set of primitive functions. We say that  $f$  calls  $g$  if  $\text{Def } f := \text{exp}$  is in the environment and  $g$  occurs in  $\text{exp}$  or if some function  $h$  occurring in  $\text{exp}$  calls  $g$ . If  $f$  calls  $f$ , then  $f$  is said to be *recursively defined*. Otherwise,  $f$  is said to have a *closed form defining equation*. Informally, we will say that  $f$  is either *recursive* or *closed form*. It is easy to see that if  $f$  calls  $g$  and  $g$  calls  $f$ , then  $g$  is recursive. In this case,  $f$  and  $g$  are said to be *mutually recursive*.

A function given by a closed form defining equation is defined uniquely by the function expression of the defining equation, but a recursively defined function is not, in general. That is, given a defining equation of the form  $\text{Def } f := Ef$ , where  $Ef$  is a function expression involving  $f$ , there may be any number of functions satisfying the equation. Each function that satisfies the equation is called a *fixed point* of that equation. We choose from among those fixed points one function, the *least fixed*

point, as the sole function denoted by the definition  $\mathbf{Def} f := Ef$ . The least fixed point always exists in this case [Tarski 1955] and can be constructed as given in Kleene's Theorem [Kleene 1952].

It may also be the case that a function  $f$  is recursive in some environment, but  $f$  does not occur in the function expression of its own defining equation. The set of functions that  $f$  calls and that also call  $f$  form a set of mutually recursive equations and are "solved simultaneously" to obtain the set of functions defined by the set of equations. Kleene's Theorem gives the solutions to recursive equations, that is, the least fixed points of the equations.

In the following theorem, the expression " $exp[d'_1, d'_2, \dots, d'_n/d_1, d_2, \dots, d_n]$ " denotes the expression identical to  $exp$  where  $d'_1$  has been substituted for all occurrences of  $d_1$  in  $exp$ ,  $d'_2$  has been substituted for all occurrences of  $d_2$  in  $exp$ , etc.

**Kleene's Theorem:** Given an environment with a finite collection of defining equations  $\mathbf{Def} f := exp_f$ ,  $\mathbf{Def} g := exp_g$ , ...,  $\mathbf{Def} h := exp_h$ , each recursively defined function is equal to the least upper bound of a sequence of functions:

$f$  is the least upper bound of the functions  $f_0, f_1, f_2, \dots$

$g$  is the least upper bound of the functions  $g_0, g_1, g_2, \dots$

⋮

$h$  is the least upper bound of the functions  $h_0, h_1, h_2, \dots$

where

$$f_0 := \bar{\perp}; g_0 := \bar{\perp}; \dots h_0 := \bar{\perp};$$

$$f_{i+1} := exp_f[f_i, g_i, \dots, h_i/f, g, \dots, h];$$

$$g_{i+1} := \text{exp}_g[f_i, g_i, \dots, h_i/f, g, \dots, h];$$

⋮

$$h_{i+1} := \text{exp}_h[f_i, g_i, \dots, h_i/f, g, \dots, h].$$

## The meanings of SFP expressions

When the semantics of a language is given denotationally, the syntactic expressions of the language are mapped by a semantic function to abstract mathematical objects, such as numbers, truth values, and functions. Frequently, common mappings are suppressed, such as mapping the numerals to numbers. We can suppress this mapping because it has been done so many times before that we are content to think of the numerals themselves as the numbers. But we must know that the symbols we use model correctly the abstract entities, or else we have not given the proper meaning to the symbols. The symbols that we use to denote the natural numbers, for example, constitute a faithful model because they satisfy Peano's Axioms for the natural numbers.

An expression of the SFP language is  $\perp$ , an atom, a prefix, a finite sequence, an infinite sequence, or an application of the form  $f(x)$ , where  $f$  is a function expression and  $x$  is an object. The meaning function  $\mu$  is defined on expressions as follows:

$$\mu(\perp) = \perp$$

$$\mu(a) = a \text{ if } a \text{ is an atom}$$

$$\mu(\langle \rangle_{i=1}^n x_i) = \langle \rangle_{i=1}^n \mu(x_i)$$

$$\mu(\langle \rangle_{i=1}^{\infty} x_i) = \langle \rangle_{i=1}^{\infty} \mu(x_i)$$

$$\mu(\langle \rangle_{i=1}^{\infty} x_i) = \langle \rangle_{i=1}^{\infty} \mu(x_i)$$

$$\mu(\top) = \top$$

$$\mu(f(x)) = \mu(f)(\mu(x))$$

$\mu(f)$  is the abstract function denoted by  $f$ .

The previous section showed how to determine which abstract function is denoted by  $f$ , depending upon whether  $f$  is a primitive function, a functional form, or a defined function, and the environment in which the expression  $f$  occurs.

We have described all of the objects and all of the functions in the SFP language, and we have defined the meaning of any of the functions applied to any of the objects. Our description of the SFP language is now complete.

### Continuity of SFP functions

We have left to show that every SFP function is continuous. First, we establish that the functional form operators preserve continuity. A functional form operator  $E$  of arity  $n$  preserves continuity if, given that  $f_1, f_2, \dots,$  and  $f_n$  are continuous functions, then the function denoted by  $E(f_1, f_2, \dots, f_n)$  is also continuous.

To show that the functional form operators preserve continuity on the functions over  $\mathbf{D}$ , we note the following. The functional form operators are defined for functions on  $\mathbf{C}$  and then extended to functional form operators for functions on  $\mathbf{D}$  by applying the resulting functional expression to each element of each chain of the elements of  $\mathbf{D}$ , as described in Definition 42. According to Theorem 44, it is sufficient to show that the functional form operators preserve monotonicity of the functions on the domain  $\mathbf{C}$ .

**Theorem 45:** The functional form operators preserve continuity.

**Proof:** We must show that for  $x, y \in \mathbf{C}$ , if  $x \sqsubseteq y$  and the function arguments to some functional form operator are monotonic, then the result of applying to  $x$  the the function that results when the functional form operator is applied to its arguments precedes the result of applying that function to  $y$ . The proofs are by cases, and we shall assume henceforth that  $y \neq \top$  to reduce the number of cases. (Examination of the definitions of the functional form operators shows that  $y = \top$  implies that the theorem is true no matter what the value of  $x$  is.) We will show the the functional form operators Condition and Left Insert preserve monotonicity; proofs for the other operators are similar.

**Condition:** (If  $p, f, g$  are monotonic, then  $p \rightarrow f;g$  is monotonic.)

Case 1:  $p(x) = \perp$ . Then  $(p \rightarrow f;g)(x) = \perp$ , which precedes everything, and thus precedes  $(p \rightarrow f;g)(y)$ .

Case 2:  $p(x) = T$ . Then  $(p \rightarrow f;g)(x) = f(x)$  and  $p(y) = T$  or  $\top$ , so  $(p \rightarrow f;g)(y) = f(y)$  or  $\top$ . Since  $f$  is monotonic and since everything precedes  $\top$ , we have that  $f(x) \sqsubseteq f(y)$  in either case.

Case 3:  $p(x) = F$ . Similar to Case 2.

Case 4:  $p(x) \neq T, F, \text{ or } \perp$ . Since  $p$  is monotonic, it follows that  $p(y) \neq T, F, \text{ or } \perp$ . Therefore,  $(p \rightarrow f;g)(x) = (p \rightarrow f;g)(y) = \top$ .

**Left Insert:** (If  $f$  is monotonic, then  $\setminus f$  is monotonic.)

Case 1:  $x = \langle x_1, \dots, x_n \rangle$  &  $n \geq 0$ . Then  $\setminus f(x) = \perp$ , and hence precedes  $\setminus f(y)$ .

Case 2:  $x = \langle x_1 \rangle$ . Then  $y = \langle y_1 \rangle$  such that  $x_1 \sqsubseteq y_1$ . Also,  $\setminus f(x) = x_1$  and  $\setminus f(y) = y_1$ , so  $\setminus f(x) \sqsubseteq \setminus f(y)$ .

Case 3:  $x = \langle x_1, x_2, \dots, x_n \rangle$  &  $n \geq 2$ . Then  $y = \langle y_1, y_2, \dots, y_n \rangle$  such that  $x_i \sqsubseteq y_i$  for  $1 \leq i \leq n$ , and  $\backslash f(x) = f(\langle \backslash f(\langle x_1, x_2, \dots, x_{n-1} \rangle), x_n \rangle)$  and  $\backslash f(y) = f(\langle \backslash f(\langle y_1, y_2, \dots, y_{n-1} \rangle), y_n \rangle)$ . By induction on the length of the sequence, we have that  $\backslash f(x) \sqsubseteq \backslash f(y)$ .

Case 4:  $x$  is an atom or  $\top$ . In this case,  $x = y$ , and the theorem is trivially true. ■

Since the SFP primitive functions are continuous and the functional form operators preserve continuity, it is easy to see that if  $f := exp$  is a closed form such that all function variable names in  $exp$  represent continuous functions, then  $f$  is continuous. In order to establish that recursively defined functions (the rest of the SFP functions) are continuous, we must first appeal to Kleene's Theorem.

If  $f$  is a recursively defined function, then Kleene's Theorem gives a sequence of approximating functions  $f_0, f_1, f_2, \dots$  such that  $f$  is the least upper bound of this sequence. It is clear that each of the approximating functions is denoted by a closed form defining equation and is therefore continuous. Then  $f$  must be continuous, since it is the least upper bound of a sequence of continuous functions. Therefore, all recursively defined SFP functions are continuous. Since all SFP functions are either closed form or recursively defined, we have shown that:

**Theorem 46:** All SFP functions are continuous.

Finally, since continuity implies monotonicity, we also have that:

**Corollary 47:** All SFP functions are monotonic.

## Summary

This chapter has been devoted to the denotational description of SFP and discussion of some of the important properties of the language and its programs (which are the functions). The section on primitive functions showed how any monotonic primitive function on Backus's FP domain  $\mathbf{O}$  can be m-extended to a continuous primitive function on  $\mathbf{C}$ , and therefore on  $\mathbf{D}$ , and that the extension mechanism given in Definition 39 produces the maximal monotonic extension. The functional forms for SFP are straightforward extensions of the functional forms for FP. The collection of SFP functions is constructed from the primitive functions, the functional forms and a defining mechanism, and it was shown that all of the SFP functions are continuous. The meanings of all SFP expressions has been denotationally described.

## Chapter 4

### Operational Semantics of SFP

In this chapter we describe an operational semantics for SFP and show that computations carried out using the operational semantics produce results equivalent to the denotationally defined results except in certain cases. The reason for differences between the denotational and operational results will be discussed.

#### The need for an operational semantics

One of the goals in the design of a programming language is an implementation such that programs written in the language may be executed correctly and with acceptable efficiency. The denotational semantics of SFP describes the results of functions applied to possibly infinite objects. To determine the result of applying a function to an object, the function, or an approximation to it in the case of recursively defined functions, is applied to a finite approximation of the input to produce an approximation to the output. To get a better approximation to the output, one need only choose (sufficiently) better approximations to the function and input. (The notion of “better” is captured by the partial order, that is,  $a$  is better than  $b$  if  $b \sqsubseteq a$ .) The actual result is the limit of these approximate results.

As a concrete example, consider the meaning, as described by the denotational semantics, of  $f(\langle 1, 1 \rangle)$  where **Def**  $f := \text{apndl} \circ [1, f \circ [+], \mathcal{I}]$ . Since  $f$  is given recursively, its meaning is the limit of a sequence of functions  $f_0, f_1, f_2, \dots$  as defined by Kleene’s theorem. The first four functions of this sequence are:

$$f_0 = \overline{1}$$

$$f_1 = \text{apndl} \circ [1, \overline{1}]$$



$$f_2 = \text{apndl} \circ [1, \text{apndl} \circ [+, \perp]]$$

$$f_3 = \text{apndl} \circ [1, \text{apndl} \circ [+, \text{apndl} \circ [+, [+, 2], \perp]]]$$

Applying each of  $f_0$ ,  $f_1$ ,  $f_2$ , and  $f_3$  to  $\langle 1, 1 \rangle$  produces  $\perp$ ,  $\langle 1 \rangle$ ,  $\langle 1, 2 \rangle$ , and  $\langle 1, 2, 3 \rangle$ , respectively. The meaning of  $f(\langle 1, 1 \rangle)$  is the limit of a sequence whose first four entries are  $\perp$ ,  $\langle 1 \rangle$ ,  $\langle 1, 2 \rangle$ , and  $\langle 1, 2, 3 \rangle$ . Informally, we describe this limit by the expression  $\langle 1, 2, 3 \dots \rangle$ .

We wish to avoid a direct implementation of the algorithm implied by Kleene's Theorem for producing the results. As seen in the example of the previous paragraph, that algorithm involves recomputing the entries of the previous approximation to the result each time a new approximation to the result is computed. The operational semantics will avoid such recomputation, for once an entry in the result has been computed, it will not be subsequently recomputed.

### The goal of an operational semantics

In general, an operational semantics will define computations that can be carried out by a machine and that will produce either the value specified by the denotational semantics or approximations, in the case of infinite results, to the denotationally defined result. We shall have to content ourselves with approximations, for a machine is not generally capable of processing or producing infinite objects. If applying a function  $f$  to an argument  $x$  produces an infinite result  $f(x)$ , then it is not possible to produce that result exactly; the most we can ask is that the system be capable of producing an arbitrarily good finite approximation to the value of  $f(x)$ . Moreover, if the argument  $x$  is itself infinite, then it is not generally possible to apply  $f$  to the argument. In that case, the most we can ask is the ability of the system to process any finite value  $\hat{x}$ , where  $\hat{x}$  is an approximation to  $x$ , and that the result produced be an approximation to  $f(x)$ . Note that in general,  $\hat{x}$  will be an approximation to many objects, both finite and infinite. Thus, the result of applying  $f$  to  $\hat{x}$  must be an approximation not only of  $f(x)$ , but also of any  $f(x')$ , where  $x'$  is one of the

objects distinct from  $x$ , but approximated by  $\hat{x}$ . We cannot even require the system to produce  $f(\hat{x})$ , because even if  $\hat{x}$  is finite,  $f(\hat{x})$  might be infinite.

We shall not be content with an operational semantics whose only guarantee is that it produces an approximation to the result, because the trivial operational semantics that produces  $\perp$  for any program and argument satisfies this guarantee. We shall see, however, that the operational semantics defined here satisfies a more substantial requirement. If the computation specified by the operational semantics halts, then either it produces the correct (finite) result or it produces something other than the correct result of  $\top$ . If the computation specified by the operational semantics does not halt, then either one of the subexpressions has a value of  $\perp$  or else the correct result is infinite (or both). In the former case, we can expect no more of an operational semantics than a non-terminating computation, and in the latter case, we will see that the result being produced by the computation is satisfactory.

### **Definition of the operational semantics for SFP**

We have used  $f(x)$  to refer to the denotational semantics of  $f$  applied to an argument  $x$  in  $\mathbf{D}$ . We will also want to refer to the result computed by use of the operational semantics, given  $f$  and an argument  $x$ , an element of  $\mathbf{C}$ . We will denote this result by  $f : x$ .

The denotational semantics for the SFP language described in the previous chapters consists of a set of expressions constructed from a domain, and a collection of functions over  $\mathbf{D}$ , together with a semantic function which maps expressions to their meanings, which are elements of  $\mathbf{D}$ . This abstract semantics admits a number of operational semantics. The one presented here consists of the domain  $\mathbf{C}$ , a collection of rewrite rules, and a parallel outermost evaluation or computation rule. The domain  $\mathbf{C}$  is used instead of  $\mathbf{D}$  because  $\mathbf{C}$  contains all the finite objects of  $\mathbf{D}$  and no infinite objects (although it does contain arbitrarily good approximations to objects in  $\mathbf{D}$ ).

## Rewrite rules

A rewrite rule consists of a left hand side and a right hand side. The left hand side is an expression possibly with conditions that govern its use. Together, these characterize the class of expressions to which the rewrite rule can be applied. The right hand side is an expression that describes a new expression that replaces the subexpression identified by the left hand side when the SFP expression is rewritten.

Consider the SFP expression  $\langle \text{length} : \langle + : \langle 4, 3 \rangle, 5 \rangle, 1 \rangle$ . One of the rewrite rules in the operational semantics is

$$\text{length} : \langle x_1, \dots, x_n \rangle, n \geq 1 \rightarrow n.$$

Here, the left hand side is the expression  $\text{length} : \langle x_1, \dots, x_n \rangle$  with the condition that  $n \geq 1$ . This left hand side matches the subexpression  $\text{length} : \langle + : \langle 4, 3 \rangle, 5 \rangle$ , where  $n = 2$ ,  $x_1 = + : \langle 4, 3 \rangle$ , and  $x_2 = 5$ . When the SFP expression  $\langle \text{length} : \langle + : \langle 4, 3 \rangle, 5 \rangle, 1 \rangle$  is rewritten by application of this rewrite rule, the result is the expression  $\langle 2, 1 \rangle$ ; the subexpression  $\text{length} : \langle + : \langle 4, 3 \rangle, 5 \rangle$  has been replaced by 2.

The collection of rewrite rules used here includes rules obtained by augmenting the definitions of the primitive functions given in Backus's Turing Award Lecture. The augmentations specify how primitive functions operate on prefixes,  $\perp$ , and  $\top$ . Since the evaluation rule is non-innermost, rewriting is not restricted to those applications whose subexpressions are constants. (This is in contrast to Backus's FP, where only applications whose proper subexpressions are constants can be rewritten.)

Additional reductions permit outermost occurrences of *apndl* to be rewritten in certain important cases. Consider the simple program of Example 2 in Chapter 6,  $\text{flat} := \text{apndl} \circ [1, \text{flat} \circ 2]$ , and apply *flat* to an argument  $x$ , where  $x$  is an infinite object. Below are several steps in the evaluation, not using the additional rules for *apndl*. The expressions  $x_1$ ,  $x_2$ , and  $x_{21}$  represent the first entry of  $x$  (i.e.,  $\mu(1 : x)$ ), the second entry of  $x$  (i.e.,  $\mu(2 : x)$ ), and the first entry of the second entry of  $x$  (i.e.,  $\mu(1 : 2 : x)$ ), respectively.

$$\begin{aligned}
\text{flat} : x &\rightarrow \text{apndl} \circ [1, \text{flat} \circ 2] : x \\
&\rightarrow \text{apndl} : [1, \text{flat} \circ 2] : x \\
&\rightarrow \text{apndl} : \langle 1 : x, \text{flat} \circ 2 : x \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{flat} : 2 : x \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{apndl} \circ [1, \text{flat} \circ 2] : 2 : x \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{apndl} : [1, \text{flat} \circ 2] : 2 : x \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{apndl} : \langle 1 : 2 : x, \text{flat} \circ 2 : 2 : x \rangle \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{apndl} : \langle 1 : x_2, \text{flat} : 2 : 2 : x \rangle \rangle \\
&\rightarrow \text{apndl} : \langle x_1, \text{apndl} : \langle x_{21}, \text{apndl} \circ [1, \text{flat} \circ 2] : 2 : 2 : x \rangle \rangle
\end{aligned}$$

Notice that with Backus's set of rewrite rules, the outermost occurrence of *apndl* cannot be reduced until the second entry of its argument is a sequence or a prefix, as required by the rewrite rules for *apndl*. But the second entry of the argument has *apndl* as its outermost operator, which cannot be reduced until the second entry of its argument is a sequence or a prefix, and so forth. This expression could continue to grow without bounds, with an occurrence of *apndl* for each new element of the result. The additional rewrite rules for *apndl* solve the problem; they allow the same initial expression to be rewritten to

$$\langle x_1, x_{21}, \sim \text{apndl} \circ [1, \text{flat} \circ 2] : 2 : 2 : x \rangle$$

where  $x_1, x_{21}$  is an unbracketed list to be appended to the left of the sequence that is the value of the expression  $\text{apndl} \circ [1, \text{flat} \circ 2] : 2 : 2 : x$

The rewrite rules fall into four categories, which are described below. The first category contains rules for primitive functions, the second for functional forms, the third for defined functions, and the last for sequences or prefixes containing  $\top$ .

1. For applications whose operators are primitive functions, the rewrite rules are formulated by 'extending' Backus's primitive functions to functions on  $\mathbf{C}$ . Note that unlike the rewrite rules for FP, the variables on the left hand side of an SFP rewrite rule can represent arbitrary well-formed expressions. Also included in this

category are the special rules, mentioned above, for *apndl*. A complete list of all the rules of this category appear in Appendix C.

Example: The application

$$distl : < 1 : < A, B >, < A, C, D >>$$

is rewritten using one of the rewrite rules for *distl*, giving

$$<< 1 : < A, B >, A >, < 1 : < A, B >, C >, < 1 : < A, B >, D >> .$$

Example: The application

$$apndl : < 1, f : x >$$

is rewritten using one of the special rules for *apndl* as

$$< 1 \sim f : x > .$$

2. Applications involving an operator built from a functional form are handled similarly in that the operational semantics is based on reductions for arguments in the domain of finite elements. A complete list of all the rules of this category also appear in Appendix C.

Example: The application

$$[f_1, f_2, f_3] : < 3, 4, [g_1, g_2] : 5 >$$

is rewritten using the rule for *construction* as

$$< f_1 : < 3, 4, [g_1, g_2] : 5 >, f_2 : < 3, 4, [g_1, g_2] : 5 >, f_3 : < 3, 4, [g_1, g_2] : 5 >> .$$

3. Applications whose operators are names of defined functions are rewritten by replacing the function symbol by its definition.

Example: Let  $\mathbf{Def} f := \mathit{apndl} \circ [\bar{1}, f]$ . Then the expression

$$\langle 1 \sim f : x \rangle$$

is rewritten as

$$\langle 1 \sim \mathit{apndl} \circ [\bar{1}, f] : x \rangle.$$

4. The fourth category contains rules to rewrite sequences or prefixes containing  $\top$  as an entry to  $\top$ . For the two rules in this category, given below, let  $x_i = \top$  for some  $i$ ,  $1 \leq i \leq n$ , in  $\langle x_1, \dots, x_n \rangle$  and  $\langle x_1, \dots, x_n \rangle$ :

$$\langle x_1 \dots x_n \rangle \rightarrow \top$$

$$\langle x_1 \dots x_n \rangle \rightarrow \top$$

Example: The expression

$$\langle 10, + : \langle 4, 3 \rangle, \top, 1233 \rangle$$

is rewritten to

$$\top.$$

### Computation rule

A computation is carried out as a sequence of steps, where each step rewrites an expression. At each step, a set of subexpressions to be rewritten is chosen and the subexpressions are rewritten to produce a new expression. Succeeding steps repeat this until no subexpressions can be rewritten. A computation rule specifies which

subexpressions are to be rewritten at each step. The choice of computation rule can affect the result.

Because sequences are  $\top$ -preserving, the denotational semantics of SFP corresponds to an innermost evaluation strategy. The semantics of FP is based on an innermost evaluation strategy for a similar reason, that is, sequences are  $\perp$ -preserving. When reducing an application by use of an innermost computation rule, the outermost level may remain an application until the reduction of all enclosed expressions is complete. That the outermost level may remain an application until completing the reduction of enclosed expressions presents no problem as long as the reduction of every expression not having a value of  $\perp$  is guaranteed to terminate, as is the case with FP. But when dealing with streams, many non-terminating computations have values other than  $\perp$ , and so an innermost evaluation strategy is not feasible. (This problem is discussed more fully in the next section, which addresses the reasons for the differences between the denotational semantics and the operational semantics of SFP.) Therefore, the computation rule for SFP must not be innermost.

We have chosen parallel outermost as the computation rule. Outermost is chosen so that non-terminating subcomputations do not necessarily prevent termination, as described above. "Parallel" is chosen to ensure, for example, that the reduction of a sequence containing two non-terminating computations as its entries makes progress on the evaluation of each of them, rather than just the left one, as with leftmost outermost, or the right one, as with rightmost outermost.

Parallel outermost evaluation is an iterative process; a single iteration involves determining a set of subexpressions to be rewritten and then rewriting them. An expression can be rewritten if and only if it matches the left hand side of a rewrite rule. A disjoint set of subexpressions to be rewritten is selected as follows. If the outermost level can be rewritten, then the entire expression is the only one selected for rewriting. If not, then each son (corresponding to each maximal proper subexpression) of the root is recursively searched in parallel for a subexpression to be rewritten. In searching any son, the search continues inward until one (maximal)

subexpression that can be rewritten is found or until it is determined that no subexpression can be rewritten. When the set of subexpressions to be rewritten has been determined, all are rewritten. Since they are disjoint, they may be rewritten in parallel or in any order. (They may even be rewritten when found.) Evaluation halts if the entire expression is searched and no subexpression that can be rewritten is found.

Suppose that a parse tree of the expression to be evaluated has been constructed. The following pseudo-code algorithm describes how to evaluate that tree according to a parallel outermost computation rule. The root of the tree corresponds to the outermost level. Initially, the root of the tree will be ‘.’; the left son will be a function expression, and the right son will be a data object. For brevity, if the root of a tree is named *root*, we denote the entire tree by {*root*}; that is, {*root*} denotes the tree whose root is *root*.

```

repeat
  eval(root)
until eval(root) = {root}.

function eval(root);
  case root of
    Left insert, Right insert, Apply-to-all, primitive function, atom: null;
    defined function name: replace name with the definition;
    <>, <|> : if any son equals ⊥ then rewrite {root} as ⊥
              else if any son equals ~ and a rewrite rule involving ~
                applies to {root}, then apply it
              else apply eval simultaneously to each son;
    ‘.’ : if a rewrite rule applies to {root}, then apply it
          else apply eval simultaneously to both sons;
  end; (* case *)
  return; {root}
end; (* function eval *)

```

The algorithm calls for the tree to be evaluated until no further rewrites can be done. Each call to ‘eval’ will result in a set of rewrites, each occurring at a single point at most along the path from a leaf to the root. Any time a set of parallel



rewrites is complete, the call to eval is finished, and the tree must be evaluated again from the root of the entire tree.

Note that some functional forms, such as *composition* and *construction* do not depend on the structure of the argument; they can always be rewritten, so that when one of them is the left son of the tree whose root is ':' and that is being evaluated, the tree is rewritable. Hence, only the functional forms, such as *apply - to - all*, that depend on the structure of the operand can ever become the root of a tree being evaluated. Therefore, they are the only ones included in the line of the case statement whose action is *null*.

An example of how the algorithm proceeds is given below. Assume the following definitions:

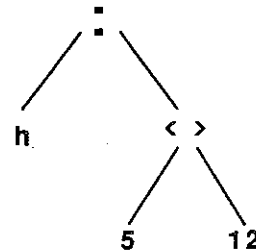
**Def**  $h := f \circ g$

**Def**  $g := [+ , -]$

**Def**  $f := *$

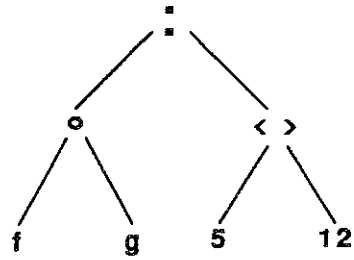
and suppose the expression  $h : < 5, 12 >$  is to be evaluated.

$h : < 5, 12 >$



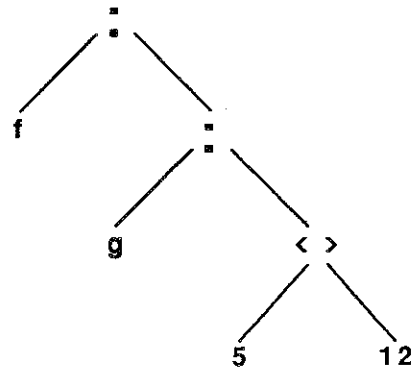
These are the initial expression and the corresponding parse tree. The root of the tree is examined to see if it can be rewritten. Since the set of rewrite rules does not contain a rule involving  $h$ , a search is made in both the left son ( $h$ ) and the right son ( $< 5, 12 >$ ). A complete search of the right son finds no rewrites, but  $h$  can be rewritten:

$f \circ g : \langle 5, 12 \rangle$



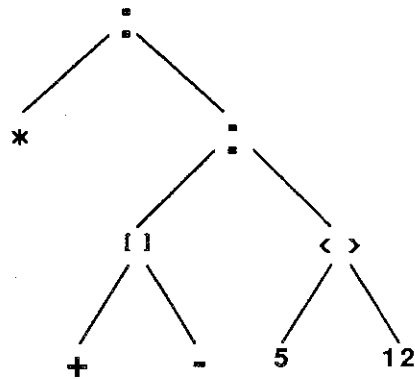
Starting at the root again, since the left son is *composition*, a rewrite can be performed and no further searching is necessary:

$f : g : \langle 5, 12 \rangle$



Again from the root, there is no rewrite rule involving  $f$ , so both the left son ( $f$ ) and the right son ( $g : \langle 5, 12 \rangle$ ) are searched. The left son can be replaced by its definition. In the right son, there is no rewrite rule involving  $g$ , so both of the right son's sons ( $g$  and  $\langle 5, 12 \rangle$ ) are searched.  $g$  can be replaced by its definition, and no rewriting can be done in  $\langle 5, 12 \rangle$ . Therefore, there are a total of two rewrites to be done in this step:

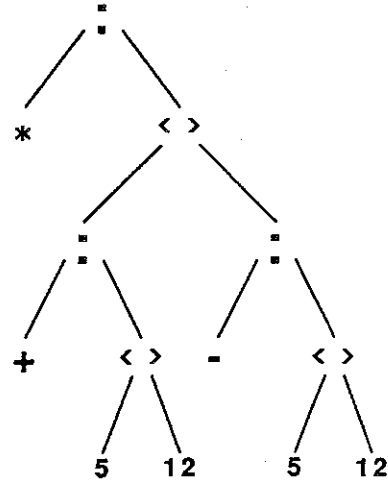
$* : [+ , -] : \langle 5, 12 \rangle$



As always, beginning at the root, none of the rules for  $*$  can be applied here, so both sons ( $*$  and  $[+ , -] : \langle 5, 12 \rangle$ ) are searched in parallel.  $*$  is a primitive function,

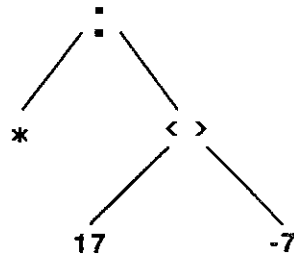
so nothing is done. The left son of  $[+, -] :< 5, 12 >$  is *construction*, which means that  $[+, -] :< 5, 12 >$  can be rewritten:

$* :< + :< 5, 12 >, - :< 5, 12 >>$



It is still not possible to apply any of the rules for  $*$ , so both sons ( $*$  and  $< + :< 5, 12 >, - :< 5, 12 >>$ ) are searched.  $*$  is a primitive function, so nothing is done. The root of  $< + :< 5, 12 >, - :< 5, 12 >>$  is  $< >$ , so each of its sons ( $+ :< 5, 12 >$  and  $- :< 5, 12 >$ ) is searched. Both are rewritable:

$* :< 17, -7 >$



At last, we can rewrite the root, since a rewrite rule for  $*$  finally applies:

-119

-119

Evaluation now terminates because  $\text{eval}(-119)$  is equal to  $-119$ .

## Connection of the operational and denotational semantics

The operational semantics does not define quite the same language as that defined by the denotational semantics. The differences stem from two sources. First, the operational semantics cannot produce an infinite result. Second, the operational semantics may ‘throw away’ a subcomputation.

The first difference arises when the result according to the denotational semantics is infinite. In this case, the operational semantics cannot produce the correct result, since it is infinite, but it will result in a non-terminating computation that is related to the infinite result. Specifically, if the computation is halted after any rewriting step and all applications are replaced by  $\perp$ , then the resulting expression is an approximation to the infinite result.

The second difference arises because non-innermost reduction will sometimes evaluate an expression without evaluating all its subexpressions; if one of the unevaluated subexpressions has a value of  $\top$ , then the operational semantics may fail to give the same result as the denotational semantics. In this case, the denotational semantics defines  $\top$  as the result, but the operational semantics may produce something other than  $\top$ . (Note that whenever  $\top$  is the result of the operational computation,  $\top$  is the denotational result, as well.)

The differences of the two semantics result from two features of SFP. One feature is that SFP has infinite objects, none of which are in the domain of the operational semantics; as a consequence, the domain of the denotational semantic function properly includes that of the operational semantic function.

The second difference between the two semantics results from the use of  $\top$  as the error object and the choice to make sequences and prefixes  $\top$ -preserving. If sequences preserve  $\top$ , then innermost evaluation is necessary to make the operational semantics correspond to the denotational semantics. But innermost evaluation would often prevent non-terminating computations from producing useful output. For example, suppose we have a program  $f \circ g$  in which  $g$  produces a stream

(from some seed) and  $f$  consumes it. Innermost evaluation would not rewrite any expression involving  $f$  until the output of  $g$  is completely computed. Since the output of the program  $g$  is infinite, innermost evaluation is totally unacceptable in this case. Consequently we have chosen to give up innermost evaluation, realizing that some computations will produce non- $\top$  objects when they should produce  $\top$ .

The discussion of the differences between the denotational semantics and the operational semantics so far has centered on why the operational semantics cannot be designed to match the denotational semantics. But it is also fair to ask why a denotational semantics that matches the operational semantics has not been given.

Recall the first difference that the denotational semantics has infinite objects while the operational semantics does not. A major goal in the design of SFP was to make possible meaningful non-terminating computations. Infinite objects form the basis for non-terminating computations. Thus, though the operational semantics cannot include infinite objects, it can approximate them arbitrarily well. Hence it is appropriate for the denotational semantics to include infinite objects which are approximated by the finite objects of the operational semantics.

The second difference was that the operational semantics sometimes fails to give  $\top$  when that is the denotational result. The reason can be traced to the decision to make sequences and prefixes  $\top$ -preserving. Now the question is, why should sequences and prefixes be  $\top$ -preserving? The answer lies in the constraints accepted for this research.

We purposed to describe streams that could be extended on the right but not anywhere else. When the argument to a function becomes better defined, or "grows," such as when it is extended on the right, the value produced by that function may grow. A constraint we accepted was that the functions should be able to produce partial results. However, one possibility is that the argument to a function may grow into something unsuitable for that function; in this case, the result must be able to indicate that an error has occurred. An easy and elegant way to handle

errors is to choose a single error value at the top of the lattice. (Another way to handle it would be with many error elements; the disadvantages of that choice will be discussed shortly.)

Even with all these constraints, it still might be possible to change the denotational semantics so that it gives the same results as the operational semantics (except for infinite objects) by ignoring some occurrences of  $\top$  when evaluating expressions. But consider how  $\top$  is introduced into an expression. It occurs when the argument to a function is inappropriate (having the wrong “shape” or type). This occurs when the programmer has written a program that does not match the argument to the function. We choose not to ignore  $\top$  here because we consider the information provided by these error messages to be of substantial import. Therefore, the denotational semantics is chosen as the semantics we really want, and the operational semantics is the best possible approximation to the denotational semantics.

Returning to the discussion of the possible ways to denote error, an alternative we have not selected is that of having many error elements. Although this seems like a worthy idea, it is not clear how it could be made to work in some cases. Consider the expression  $trans : \langle \langle 1, 2 \rangle, \langle 3, 4 \rangle \rangle$ . One of the constraints is that we must put out partial results. Thus the value of this expression is  $\langle \langle 1, 3 \rangle, \langle 2, 4 \rangle \rangle$ . The argument  $\langle \langle 1, 2 \rangle, \langle 3, 4 \rangle \rangle$  could grow into a number of elements, including  $\langle \langle 1, 2 \rangle, \langle 3, 4, 5 \rangle \rangle$  and  $\langle \langle 1, 2 \rangle, \langle 3, 4 \rangle, 5 \rangle$  (both of which could grow into  $\langle \langle 1, 2 \rangle, \langle 3, 4, 5 \rangle, 5 \rangle$ ). Both of these intermediate elements are inappropriate arguments for  $trans$ ; thus some error objects must be defined to be the result of  $trans : \langle \langle 1, 2 \rangle, \langle 3, 4, 5 \rangle \rangle$  and  $trans : \langle \langle 1, 2 \rangle, \langle 3, 4 \rangle, 5 \rangle$ . These error objects must be preceded by  $\langle \langle 1, 3 \rangle, \langle 2, 4 \rangle \rangle$  and both must precede whatever is chosen as the value of  $trans : \langle \langle 1, 2 \rangle, \langle 3, 4, 5 \rangle, 5 \rangle$ . Clearly any error scheme capable of handling these myriad cases will be complex, and we have not addressed the problem in this research.

The remainder of this chapter is devoted to proving our claim that the operational semantics is equivalent to the denotational semantics except as noted. We classify rewrite rules as *application-preserving*, and show that a rewrite rule that is application-preserving may be applied at any level (to any subexpression) of an expression, and hence outermost, without changing the meaning of the expression. If a rewrite rule that is not application-preserving is applied at some level other than innermost, we show that the meaning of the resulting expression may be changed as the result of some subexpressions containing applications having been discarded, possibly eliminating  $\top$ .

Let the expression  $\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_m)$ , where  $m \leq n$ , represent a rewrite rule;  $\mathcal{F}$  and  $\mathcal{G}$  are syntactic expressions and  $e_1, e_2, \dots, e_n$  are the parameters (subexpressions) of the expressions  $\mathcal{F}$  and  $\mathcal{G}$ . Substitution instances of the parameters will be denoted by capital letters:  $E_1, E_2, \dots$ . Thus  $(\mathcal{E}, \{\mathcal{F}(E_1, E_2, \dots, E_n)\})$  represents an expression  $\mathcal{E}$  that contains an instance of a rewrite rule's left hand side  $\mathcal{F}(E_1, E_2, \dots, E_n)$  that can be reduced to  $\mathcal{G}(E_1, E_2, \dots, E_m)$  giving a new expression denoted by  $(\mathcal{E}, \{\mathcal{G}(E_1, E_2, \dots, E_m)\})$ .

We wish to discuss the meanings of the rewrite rule expressions, and so we observe that the meaning function  $\mu$  defined in Chapter 3 implies that the meaning of an expression is equal to the meaning of that expression whose subexpressions have been replaced by their meanings. That is,

$$\mu(\mathcal{F}(e_1, e_2, \dots, e_n)) = \mu(\mathcal{F}(\mu(e_1), \mu(e_2), \dots, \mu(e_n))),$$

$$\mu(\mathcal{G}(e_1, e_2, \dots, e_n)) = \mu(\mathcal{G}(\mu(e_1), \mu(e_2), \dots, \mu(e_n))),$$

$$\mu(\mathcal{E}, \{\mathcal{F}(E_1, E_2, \dots, E_n)\}) = \mu(\mathcal{E}, \mu(\{\mathcal{F}(E_1, E_2, \dots, E_n)\})),$$

and so forth.

**Definition 48:** A rewrite rule is *application-preserving* if every expression parameter on the left appears on the right, that is,

$$\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_n)$$

This property ensures that no application is discarded when the rewrite rule is applied. An example of an application-preserving rewrite rule is *distl* :  $\langle x, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \langle \langle x, y_1 \rangle, \dots, \langle x, y_n \rangle \rangle$ . An example of a rewrite rule that is not application-preserving is *tail* :  $\langle x_1, x_2, \dots, x_n \rangle \rightarrow \langle x_2, \dots, x_n \rangle$ . Here the parameter  $x_1$  appears on the left hand side but not on the right hand side.

A function definition is application-preserving if it consists only of application-preserving rewrite rules. Informally, we will refer to the function itself as being application-preserving if its definition is application-preserving.

Theorem 51 asserts that an application-preserving rewrite rule can be applied to an expression resulting in a new expression whose meaning is the same as the original expression. Before proving that, we will need to show that each rewrite rule correctly implements the denotational semantics when the parameters are constants, that is, when the subexpression being rewritten contains no applications. This must be done on a case-by-case basis for each primitive function.

We begin in Lemma 49 with a proof that the denotational definitions of the primitive functions are given in Appendix B. We then show in Corollary 50 that the rewrite rules in Appendix C implement the definitions in Appendix B for constant arguments. This connects the two semantics for any rewriting done at the innermost level. Following that, a series of theorems finishes the connection between the two semantics by describing what happens when a rewrite rule is applied non-innermost.

**Lemma 49:** The definitions given in Appendix B are the SFP primitive functions given by the denotational semantics.

**Proof:** The proof consists of showing that each of the functions defined in Appendix B is the same as the m-extension, as specified by Definitions 36, 39, and 42, of the Turing Award Lecture primitive function of the same name. The domain of all of the functions given in Appendix B is **D**; though



each of the elements of  $\mathbf{D}$  is an equivalence class of chains, we will not treat them as such except when necessary, and even then we will examine a single chain rather than the whole class. This is possible in light of the result of Theorem 30, which says that it suffices to consider the characteristics of a single chain. Furthermore, by observing that for each finite element  $x$ , one of the elements of the equivalence class is  $x \sqsubseteq x \sqsubseteq x \dots$ , it is clear that it suffices to examine the properties of  $x$  alone in order to draw conclusions about the entire equivalence class. Hence it is correct to think of finite elements as elements of  $\mathbf{C}$  rather than as equivalence classes of chains.

Below we give the proof required for the primitive function *tail*. Proofs for other primitive functions are similar. Some of the proofs are tedious; indeed, construction of the definitions in Appendix B is a non-trivial task.

To show that the definition of *tail* given in Appendix B is the denotational defined function *tail*, we will show that for each element  $x$  of  $\mathbf{D}$ , the given definition of *tail* matches  $x$  to the proper object. The proof is by cases of the elements of  $\mathbf{D}$ .

For convenience, in each case, we will denote the denotationally defined result by  $tail'(x)$  and the result specified by the definition in Appendix B by  $tl(x)$ . Recall that *tail'* is the extension to  $\mathbf{C}$  of the function *tail* defined on  $\mathbf{B}$ .

Case 1:  $x = \perp$ . Then

$$\begin{aligned} tail'(\perp) &= glb\{tail(y) \mid \perp \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\ &= glb\{tail(y) \mid y \in \mathbf{B}\} \\ &= \perp, \text{ since } \perp \in \mathbf{B} \ \& \ tail(\perp) = \perp \\ &= tl(\perp) \end{aligned}$$

Case 2:  $x$  is an atom. Then

$$\begin{aligned}
tail'(x) &= glb\{tail(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y = x \text{ or } y = \top\} \\
&= glb\{tail(x), tail(\top)\} \\
&= glb\{\top\} \\
&= \top \\
&= tl(\top)
\end{aligned}$$

Case 3:  $x = \langle \downarrow \rangle$ . Then

$$\begin{aligned}
tail'(\langle \downarrow \rangle) &= glb\{tail(y) \mid \langle \downarrow \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y \text{ is a sequence} \} \\
&= glb\{tail(\langle \rangle), tail(\langle 1, 5 \rangle), tail(\langle 4, 2, 3, 7, 2 \rangle), \dots\} \\
&= glb\{\top, \langle 5 \rangle, \langle 2, 3, 7, 2 \rangle, \dots\} \\
&= \langle \downarrow \rangle \\
&= tl(\langle \downarrow \rangle)
\end{aligned}$$

Case 4:  $x = \langle \rangle$ . Then

$$\begin{aligned}
tail'(\langle \rangle) &= glb\{tail(y) \mid \langle \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y = \langle \rangle \text{ or } y = \top\} \\
&= glb\{tail(\langle \rangle), tail(\top)\} \\
&= glb\{\top\} \\
&= \top \\
&= tl(\langle \rangle)
\end{aligned}$$

Case 5:  $x = \langle x_1, \dots, x_n \downarrow \rangle$ ,  $x_i \in \mathbf{D}$ ,  $1 \leq n < \infty$ .

$$\begin{aligned}
tail'(\langle x_1, \dots, x_n \rangle) &= glb\{tail(y) \mid \langle x_1, \dots, x_n \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y \text{ is a sequence of length } n \\
&\quad \text{or greater and has as its first } n \text{ entries } y_1, \dots, y_n \\
&\quad \text{where for each } i \ x_i \sqsubseteq y_i\} \\
&= glb\{z \mid z \text{ is a sequence of length } n - 1 \text{ or} \\
&\quad \text{greater and has as its first } n - 1 \text{ entries} \\
&\quad y_2, \dots, y_n\} \\
&= \langle x_1, \dots, x_n \rangle \\
&= tl(\langle x_1, \dots, x_n \rangle)
\end{aligned}$$

Case 6:  $x = \langle x_1, \dots, x_n \rangle$ ,  $x_i \in \mathbf{D}$ ,  $1 \leq n \leq \infty$ .

$$\begin{aligned}
tail'(\langle x_1, \dots, x_n \rangle) &= glb\{tail(y) \mid \langle x_1, \dots, x_n \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y = \langle y_1, \dots, y_n \rangle, \ x_i \sqsubseteq y_i, \ \text{or } y = \top\} \\
&= glb\{tail(\langle y_1, \dots, y_n \rangle), \ tail(\top)\} \\
&= glb\{\langle y_2, \dots, y_n \rangle, \ \top\} \\
&= \langle x_2, \dots, x_n \rangle \\
&= tl(\langle x_1, \dots, x_n \rangle)
\end{aligned}$$

Case 7:  $x = \top$ . Then

$$\begin{aligned}
tail'(\top) &= glb\{tail(y) \mid \top \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= glb\{tail(y) \mid y = \top\} \\
&= glb\{tail(\top)\} \\
&= glb\{\top\} \\
&= \top \\
&= tl(\top)
\end{aligned}$$

■

**Corollary 50:** The rewrite rules in Appendix C implement the definitions in Appendix B for all constant arguments in the rules of Appendix C.

**Proof:** The proof consists of examining the rewrite rules for each function, noting that except for the special rules associated with *apndl*, they are syntactically identical to the semantic functions given in Appendix B except that the infinite objects are missing from Appendix C and that the last line has been changed to map only constant expressions to  $\top$ .

The infinite objects are removed because the domain of the denotational semantics is  $\mathbf{D}$ , but the domain of the operational semantics is  $\mathbf{C}$ . Infinite objects are the only objects in  $\mathbf{D}$  that are not in  $\mathbf{C}$ .

The reason for changing the last line is a bit more subtle. The last line in the definition of the denotational semantic function maps any argument not matched by a previous line to  $\top$ . In the operational semantics, we do not wish to map everything that does not match a previous line to  $\top$ , only those expressions that are constants and do not match a previous line. If we mapped everything else to  $\top$ , then each time a primitive function was encountered during evaluation, some rewrite rule would always match, since the last one would by default. This would cause some expressions to be mapped inappropriately to  $\top$ . This is the case, for example, with  $distl : \langle 1, [+,-] : \langle 2, 4 \rangle \rangle$ , where we are attempting to rewrite *distl*. We do not want to map the entire expression to  $\top$ ; none of the regular rules for *distl* match yet because the argument is not sufficiently developed to have the proper structure, a pair whose second entry is a sequence or prefix, for *distl*. However, when the inner expression  $[+,-] : \langle 2, 4 \rangle$  is evaluated, we will have the proper structure so that one of the regular rules for *distl* will then match the expression.

The preceding discussion takes care of all the rewrite rules except for the special rules associated with *apndl*. Two of the rules,  $apndl : \langle x, y \rangle \rightarrow \langle x \sim y \rangle$  and  $apndl : \langle x, y \rangle \rightarrow \langle x \sim y \rangle$ , replace *apndl* with an infix operator ' $\sim$ .' Examination of the other rules suffices to show that ' $\sim$ ' is equivalent in function to the prefix form of *apndl*. The parameter *y* may

eventually be revealed to be  $\perp$ , a sequence or prefix, an atom, or  $\top$ . Each one of these cases is covered, and the proper thing is done. For example, suppose  $apndl : \langle x, y \rangle$  is replaced by  $\langle x \sim y \rangle$  and through evaluation,  $y$  is determined to have a value of  $\top$ . Then replacing  $\langle x \sim \top \rangle$  with  $\top$  is the right thing to do, since we would also replace  $apndl : \langle x, \top \rangle$  with  $\top$ . Thus the meaning of the expression  $apndl \langle x, y \rangle$  has not been changed by replacing it with  $\langle x \sim y \rangle$ , as long as there is a suitable rule for  $\langle x \sim y \rangle$  regardless of what  $y$  evaluates to. ■

To review, Lemma 49 and Corollary 50 have shown that the rewrite rules in Appendix C correctly implement the denotational semantics for constant arguments. Therefore, the rules correctly implement the denotational semantics for innermost evaluation of terminating computations. But we are interested in non-innermost evaluation and non-terminating computations. Theorem 51 and Lemma 52 show that application-preserving rewrite rules can be applied at any level without changing the meaning of the expression.

**Theorem 51:** Let  $\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_n)$  be an application-preserving rewrite rule. Then

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \mu(\mathcal{G}(E_1, E_2, \dots, E_n)).$$

**Proof:** If the rewrite rule  $\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_n)$  correctly implements the denotational semantics, then

$$\mu(\mathcal{F}(\mu(E_1), \mu(E_2), \dots, \mu(E_n))) = \mu(\mathcal{G}(\mu(E_1), \mu(E_2), \dots, \mu(E_n))),$$

since each  $\mu(E_i)$  is a constant. From this equation we can conclude that

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \mu(\mathcal{G}(E_1, E_2, \dots, E_n)).$$

■

The next lemma states that the level at which the reduction of an application-preserving function within an expression is done does not change the expression's value.

**Lemma 52:** Suppose  $f$  is application-preserving and  $\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_n)$  is a rewrite rule for  $f$ . Then

$$\mu(\mathcal{E}, \{\mathcal{F}(E_1, E_2, \dots, E_n)\}) = \mu(\mathcal{E}, \{\mathcal{G}(E_1, E_2, \dots, E_n)\}).$$

**Proof:**

$$\begin{aligned} \mu(\mathcal{E}, \{\mathcal{F}(E_1, E_2, \dots, E_n)\}) &= \mu(\mathcal{E}, (\mu\{\mathcal{F}(E_1, E_2, \dots, E_n)\})) \\ &= \mu(\mathcal{E}, (\mu\{\mathcal{G}(E_1, E_2, \dots, E_n)\})) \text{ by Theorem 51} \\ &= \mu(\mathcal{E}, \{\mathcal{G}(E_1, E_2, \dots, E_n)\}). \end{aligned}$$

■

It is clear from Lemma 52 that if  $\mathcal{E}$  has multiple, non-intersecting applications that can be rewritten by application preserving rules, then the meaning of  $\mathcal{E}$  does not change when all of these have been rewritten simultaneously, since they do not intersect and could be rewritten sequentially in any order without changing the meaning of the expression.

Theorem 53 states that if a non-application-preserving rewrite rule is applied, then the meaning of the expression may be changed from  $\top$  to something else because some subexpression whose meaning is  $\top$  has been discarded by application of that rewrite rule.

**Theorem 53:** Suppose  $f$  is not application-preserving, that is, one of the rewrite rules for  $f$  is  $\mathcal{F}(e_1, e_2, \dots, e_n) \rightarrow \mathcal{G}(e_1, e_2, \dots, e_k)$  where  $k < n$ . Then  $[\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) \neq \mu(\mathcal{G}(E_1, E_2, \dots, E_k))]$  implies  $[\mu(E_j) = \top \text{ for some } j, k+1 \leq j \leq n]$ , and  $\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \top$ .

**Proof:**

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \mu(\mathcal{F}(\mu(E_1), \mu(E_2), \dots, \mu(E_n)))$$

and

$$\mu(\mathcal{G}(E_1, E_2, \dots, E_k)) = \mu(\mathcal{G}(\mu(E_1), \mu(E_2), \dots, \mu(E_k))).$$

If  $\mu(E_i) \neq \top$  for  $1 \leq i \leq n$ , then

$$\mu(\mathcal{F}(\mu(E_1), \mu(E_2), \dots, \mu(E_n))) = \mu(\mathcal{G}(\mu(E_1), \mu(E_2), \dots, \mu(E_k))),$$

since  $\mu(E_i)$  is a constant and the rewrite rules correctly implement the denotational semantics for constants by Lemma 49.

So, if  $\mu(E_i) \neq \top$  for all  $i$ ,  $1 \leq i \leq n$ , it follows that

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \mu(\mathcal{G}(E_1, E_2, \dots, E_k)).$$

But by hypothesis,

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) \neq \mu(\mathcal{G}(E_1, E_2, \dots, E_k)).$$

So,  $\mu(E_j) = \top$  for some  $j$ ,  $1 \leq j \leq n$ . We argue that  $j$  cannot be between 1 and  $k$ , inclusive, or else both expressions would have the value  $\top$  and thus be equal. Therefore, we must have that  $\mu(E_j) = \top$  for some  $j$ , where  $k+1 \leq j \leq n$ . Furthermore, since one of the subexpressions of  $(\mathcal{F}(E_1, E_2, \dots, E_n))$  is  $\top$ , then

$$\mu(\mathcal{F}(E_1, E_2, \dots, E_n)) = \top.$$

■

Finally, Theorem 54 states the connection between the denotational semantics and the operational semantics. The meaning,  $\mu(\mathcal{E}_0)$ , as described by the denotational semantics, of an expression  $\mathcal{E}_0$  is produced as a constant,  $\mathcal{E}_m$ , by the operational semantics if the computation terminates and  $\mu(\mathcal{E}_0) \neq \top$ .

**Theorem 54:** If  $\mathcal{E}_0$  can be reduced in  $m$  parallel outermost steps to a constant expression  $\mathcal{E}_m$ , that is,

$$\mathcal{E}_0 \rightarrow \mathcal{E}_1 \rightarrow \mathcal{E}_2 \rightarrow \dots \rightarrow \mathcal{E}_m$$

and if  $\mu(\mathcal{E}_0) \neq \top$ , then  $\mu(\mathcal{E}_0) = \mathcal{E}_m$ .

**Proof:** By induction. Basis step:  $m = 1$ . Then there is only one level at which rewriting can be done, that is, all proper subexpressions of all applications are constants. Hence, parallel outermost rewrites all applications and produces an expression with no applications, which is the correct value.

By the induction hypothesis, if  $\mu(\mathcal{E}_1) \neq \top$ , then  $\mu(\mathcal{E}_1) = \mathcal{E}_m$ .

If  $\mu(\mathcal{E}_0) \neq \mu(\mathcal{E}_1)$ , then by Theorem 53 a non-application-preserving function was applied in the step  $\mathcal{E}_0 \rightarrow \mathcal{E}_1$ , and  $\mu(\mathcal{E}_0) = \top$ . This contradicts the hypothesis, so we must have that  $\mu(\mathcal{E}_0) = \mu(\mathcal{E}_1)$ , and so  $\mu(\mathcal{E}_0) = \mathcal{E}_m$ . ■

We have shown that for functions applied to finite arguments and having finite results, the operational semantics either produces the correct result or fails to produce  $\top$  when that is the correct result. Our quest is a set of computations that consume or produce infinite objects. Since we have not given a finite representation to all infinite objects, such computations may be non-terminating. Theorem 54 does not address the validity of the results of non-terminating computations.

Let us consider the result of a non-terminating computation. By definition, a non-terminating computation never produces a constant expression; at any point the expression still contains applications. Our goal is to show that the expression approximates the correct result in some way. One way to obtain a constant that is an approximation to the result is to replace all unfinished computations by  $\perp$ . However, it is not always possible to simply replace the applications by  $\perp$ , since some expressions contain occurrences of ‘ $\sim$ .’ These expressions require some additional rewriting. We do not choose to replace the minimal well-formed expression containing ‘ $\sim$ ’ with  $\perp$ , since that would result in some computations being replaced with  $\perp$  regardless of how many steps of the evaluation had already occurred. In other words, we desire an approximation that retains any constants already computed but does not contain any applications. The following algorithm describes



a way in which the expression being evaluated can be altered to produce a finite approximation to the correct result.

```
while applications remain
  replace all innermost applications with  $\perp$ ;
  evaluate;
until expression is a constant.

procedure evaluate;
  while outermost rewritable subexpressions remain
    perform one step of parallel outermost evaluation except
      that no defined functions are replaced by their definitions
      and no while loops are rewritten;
  return;
end; (* procedure evaluate *)
```

Demonstrating that the algorithm produces the desired result requires showing that the algorithm always terminates and that the result is an approximation to the correct result.

To see that the algorithm always terminates, we observe that the outer *while* loop terminates if the nesting level of applications always decreases each time through and if the procedure *evaluate* always terminates. Initially, the nesting level of applications may increase, since rewriting occurrences of either form of *insert* may increase the nesting level. However, since the expression is composed of a finite number of symbols, we know that it contains a finite number of occurrences of *insert*. No new occurrences of *insert* can be introduced, because defined functions cannot be replaced with their definitions. Therefore, when all occurrences of *insert* have either been rewritten or replaced by *bot*, the nesting level of applications will decrease each time through, since the innermost applications are replaced by  $\perp$ , reducing the nesting level by 1.

The procedure *evaluate* always terminates if the number of rewritable subexpressions decreases each time through. Only the following actions increase the number of applications in an expression: rewriting *insert*, *construction*, *apply-to-all*,

or *while* or replacing a defined function with its definition. As with nesting levels and *insert*, the total number of applications than can be introduced by rewriting *insert*, *construction*, and *apply – to – all* is finite; once all existing occurrences are rewritten no new ones can be introduced. The other application-increasing actions, rewriting *while* and replacing a defined function with its definition, are not allowed.

We have left to show that the result produced by the algorithm is an approximation to the denotationally defined value. Replacing a subexpression by  $\perp$  produces an approximation to the entire expression. Otherwise, applications are rewritten to according to their rewrite rules.

### Summary

An operational semantics for SFP consisting of a set of rewrite rules and an computation rule has been described. We have shown that it produces the same result as the denotational semantics, with some exceptions. One exception arises because machines, and therefore operational semantics, cannot produce the entire result if that result is infinite. However, in this case the operational semantics describes a non-terminating sequence of rewrites that produces an expression that can be viewed at any point in time as an approximation to an infinite result. The other exception occurs when a non-application-preserving function is applied at some level other than innermost. In this case, it is possible (though not inevitable) that the operational semantics may fail to produce the correct result of  $\top$ .

## Chapter 5

### Algebra

An algebraic law relates two algebraic expressions. A program in FP is an algebraic expression, and it is sometimes desirable to replace a program with another that is equivalent to it. For example, one use of algebra is to give a proof of equivalence of two programs. In the Turing Award Lecture [Backus 1978], Backus gives a proof that two different matrix multiplication programs are equivalent. Algebra can also be used to improve efficiency. Kieburtz and Shultis [Kieburtz & Shultis 1981] show how to transform some inefficient FP programs to more efficient ones through an algebraic system. Finally, algebra can be used to demonstrate that a particular program has a particular property. Example 8 of Chapter 6 illustrates this use of algebra.

The algebra associated with FP programs is one of the language's great strengths. Traditional languages have no such algebras associated with them, and consequently, the important concerns of program equivalence, program optimization, and program verification are much more difficult to address. These concerns have become increasingly more important and difficult to handle as software has become more complex and more crucial to modern technology. It is clear that the software of the future, indeed, that of today, needs much more powerful tools for reasoning about programs. The FP algebra exhibits the kind of strength that is needed.

Because of the importance of the algebra of FP, any modification of the language should preserve the algebra. The algebra survives the extension of FP to SFP with

only minor changes. Since the denotational semantics and the operational semantics of SFP define two slightly different languages, the laws are slightly different, depending on which semantics is used.

Below we give algebraic laws without proof for the denotational semantics of SFP, and when they differ from either of the other two, both the corresponding law for the operational semantics of SFP and the original FP law. The primes and stars, indicating the extension of functions, have been omitted. The laws given are taken from the Turing Award Lecture, and the same numbering system is used here as there. The list of laws is illustrative of the kinds of laws that might be used in practice and is not exhaustive.

In the following, the notation  $f \& g$  is equivalent to  $and \circ [f, g]$ , and the function “pair” is defined as  $atom \rightarrow \bar{F}; eq \circ [length, \bar{2}]$ .

### Algebraic Laws

- I      Composition and construction
- I.1     $[f_1, \dots, f_n] \circ g \equiv [f_1 \circ g, \dots, f_n \circ g]$
- I.2     $\alpha f \circ [g_1, \dots, g_n] \equiv [f \circ g_1, \dots, f \circ g_n]$
- I.3     $/f \circ [g_1, \dots, g_n]$   
 $\equiv f \circ [g_1, /f \circ [g_2, \dots, g_n]]$  when  $n \geq 2$   
 $\equiv f \circ [g_1, f \circ [g_2, \dots, f \circ [g_{n-1}, g_n] \dots]]$   
 $/f \circ [g] \equiv g$
- I.4     $f \circ [\bar{x}, g] \equiv (bu \ f \ x) \circ g$
- I.5     $1 \circ [f_1, \dots, f_n] \geq f_1$   
 $s \circ [f_1, \dots, f_s, \dots, f_n] \geq f_s$  for any selector  $s, s \leq n$   
defined  $\circ f_i$  (for all  $i \neq s, 1 \leq i \leq n$ )  $\rightarrow s \circ [f_1, \dots, f_n] \equiv f_s$

### Operational semantics:

- I.5     $1 \circ [f_1, \dots, f_n] \equiv f_1$   
 $s \circ [f_1, \dots, f_s, \dots, f_n] \equiv f_s$  for any selector  $s, s \leq n$

**Original:**

- I.5  $1 \circ [f_1, \dots, f_n] \leq f_1$   
 $s \circ [f_1, \dots, f_s, \dots, f_n] \leq f_s$  for any selector  $s, s \leq n$   
 defined  $\circ f_i$  (for all  $i \neq s, 1 \leq i \leq n$ )  $\rightarrow\rightarrow s \circ [f_1, \dots, f_n] \equiv f_s$
- I.5.1  $[f_1 \circ 1, \dots, f_n \circ n] \circ [g_1, \dots, g_n] \equiv [f_1 \circ g_1, \dots, f_n \circ g_n]$
- I.6  $tl \circ [f_1] \geq \bar{\phi}$  and  
 $tl \circ [f_1, \dots, f_n] \geq [f_2, \dots, f_n]$  for  $n \geq 2$   
 defined  $\circ f_1 \rightarrow\rightarrow tl \circ [f_1] \equiv \bar{\phi}$   
 and  $tl \circ [f_1, \dots, f_n] \equiv [f_2, \dots, f_n]$  for  $n \geq 2$

**Operational Semantics:**

- I.6  $tl \circ [f_1] \equiv \bar{\phi}$  and  
 $tl \circ [f_1, \dots, f_n] \equiv [f_2, \dots, f_n]$  for  $n \geq 2$

**Original:**

- I.6  $tl \circ [f_1] \leq \bar{\phi}$  and  
 $tl \circ [f_1, \dots, f_n] \leq [f_2, \dots, f_n]$  for  $n \geq 2$   
 defined  $\circ f_1 \rightarrow\rightarrow tl \circ [f_1] \equiv \bar{\phi}$   
 and  $tl \circ [f_1, \dots, f_n] \equiv [f_2, \dots, f_n]$  for  $n \geq 2$
- I.7  $distl \circ [f, [g_1, \dots, g_n]] \equiv [[f, g_1], \dots, [f, g_n]]$   
 defined  $\circ f \rightarrow\rightarrow distl \circ [f, \bar{\phi}] \equiv \bar{\phi}$   
 The analogous law holds for *distr*.

**Operational semantics:**

- I.7  $distl \circ [f, [g_1, \dots, g_n]] \equiv [[f, g_1], \dots, [f, g_n]]$   
 $distl \circ [f, \bar{\phi}] \equiv \bar{\phi}$   
 The analogous law holds for *distr*.

**Original:**

- I.7  $distl \circ [f, [g_1, \dots, g_n]] \equiv [[f, g_1], \dots, [f, g_n]]$   
 defined  $\circ f \rightarrow\rightarrow distl \circ [f, \bar{\phi}] \equiv \bar{\phi}$   
 The analogous law holds for *distr*.

$$\text{I.8} \quad \text{apndl} \circ [f, [g_1, \dots, g_n]] \equiv [f, g_1, \dots, g_n]$$

$$\text{nullog} \rightarrow \rightarrow \text{apndl} \circ [f, g] \equiv [f]$$

And so on for apndr, reverse, rotl, etc.

$$\text{I.9} \quad [\dots, \bar{\top}, \dots] \equiv \bar{\top}$$

### Operational Semantics:

I.9 no corresponding law

### Original:

$$\text{I.9} \quad [\dots, \bar{\perp}, \dots] \equiv \bar{\perp}$$

$$\text{I.10} \quad \text{apndl} \circ [f \circ g, \alpha f \circ h] \equiv \alpha f \circ \text{apndl} \circ [g, h]$$

$$\text{I.11} \quad \text{pair \& notonnull} \circ 1 \rightarrow \rightarrow$$

$$\text{apndl} \circ [[1 \circ 1, 2], \text{distr} \circ [tl \circ 1, 2]] \equiv \text{distr}$$

II Composition and condition (right associated parentheses omitted)

$$\text{II.1} \quad (p \rightarrow f; g) \circ h \equiv (p \circ h) \rightarrow f \circ h; g \circ h$$

$$\text{II.2} \quad h \circ (p \rightarrow f; g) \equiv p \rightarrow h \circ f; h \circ g$$

$$\text{II.3} \quad \text{or} \circ [q, \text{not} \circ q] \rightarrow \rightarrow \text{and} \circ [p, q] \rightarrow f;$$

$$\text{and} \circ [p, \text{not} \circ q] \rightarrow g; h \equiv p \rightarrow (q \rightarrow f; g); h$$

$$\text{II.3.1} \quad p \rightarrow (p \rightarrow f; g); h \equiv p \rightarrow f; h$$

III Composition and miscellaneous

$$\text{III.1} \quad \bar{x} \circ f \geq \bar{x}$$

$$\text{defined} \circ f \rightarrow \rightarrow \bar{x} \circ f \equiv \bar{x}$$

### Operational semantics:

$$\text{III.1} \quad \bar{x} \circ f \equiv \bar{x}$$

**Original:**

III.1  $\bar{x} \circ f \leq \bar{x}$   
defined  $\circ f \rightarrow \bar{x} \circ f \equiv \bar{x}$

III.1.1  $\bar{1} \circ f \equiv f \circ \bar{1} \equiv \bar{1}$

**Operational Semantics:**

III.1.1 (Same as denotational)

**Original:**

III.1.1  $\bar{1} \circ f \equiv f \circ \bar{1} \equiv \bar{1}$

III.2  $f \circ \text{id} \equiv \text{id} \circ f \equiv f$

III.3  $\text{pair} \ \& \ \text{not} \circ \text{eq} \circ [\bar{1}, 2] \ \& \ \text{not} \circ \text{eq} \circ [\overline{\langle \rangle}, 2] \rightarrow 1 \circ \text{distr} \equiv [1 \circ 1, 2]$  also:  
 $\text{pair} \rightarrow 1 \circ \text{tl} \equiv 2$  etc.

**Operational Semantics:**

III.3 (Same as denotational)

**Original:**

III.3  $\text{pair} \rightarrow 1 \circ \text{distr} \equiv [1 \circ 1, 2]$  also:  
 $\text{pair} \rightarrow 1 \circ \text{tl} \equiv 2$  etc.

III.4  $\alpha(f \circ g) \equiv \alpha f \circ \alpha g$

III.5  $\text{null} \circ g \rightarrow \alpha f \circ g \equiv \bar{\phi}$

IV Condition and construction

IV.1  $[f_1, \dots, (p \rightarrow g; h), \dots, f_n]$   
 $\equiv p \rightarrow [f_1, \dots, g, \dots, f_n]; [f_1, \dots, h, \dots, f_n]$

IV.1.1  $[f_1, \dots, (p_1 \rightarrow g_1; \dots; p_n \rightarrow g_n; h), \dots, f_m]$   
 $\equiv p_1 \rightarrow [f_1, \dots, g_1, \dots, f_m];$   
 $\dots; p_n \rightarrow [f_1, \dots, g_n, \dots, f_m]; [f_1, \dots, h, \dots, f_m]$

## Discussion

Very few changes to the laws occur, and the changes are minor. The laws that are changed are I.5, I.6, I.7, I.9, III.1, III.1.1, and III.3. Sometimes the changes are different for the two semantics (I.5, I.6, I.7, I.9, and III.1). The majority of the laws are unchanged.

For the denotational semantics, most of the laws are the same as the original laws, although in some cases, the proofs must be changed slightly (for example, with I.10:  $apndl \circ [f \circ g, \alpha f \circ h] \equiv \alpha f \circ apndl \circ [g, h]$ ). Laws containing an inequality, such as I.5 ( $1 \circ [f_1, \dots, f_n] \leq f_1$ ), are the same except that the direction of the inequality is reversed. Law I.9 differs in that  $\top$  appears in the new laws where  $\perp$  appears in the old law. Finally, some laws, such as I.6 ( $defined \circ f_1 \rightarrow tail \circ [f_1] \equiv \bar{\phi}$ ), contain as part of a pre-condition a function named “*defined*” which is equivalent to  $\overline{true}$ . Although these laws are the same in the SFP algebra, it is perhaps misleading here to use the name “*defined*,” since  $\perp$  is the totally undefined element but  $defined:\perp = true$ . Perhaps a better name would be “*not.top*”.

The laws for the operational semantics differ from the laws of the denotational semantics because the operational semantics specifies outermost evaluation. Thus the inequalities in laws I.5, I.6, I.7 and III.1 are strengthened to equality when an outermost computation rule is used. This kind of difference between the denotational semantics law and the operational semantics law accounts for every difference between the two save one, law I.9. Here, the denotational semantics law is  $[..., \bar{\top}, ...] \equiv \bar{\top}$ , but there is no corresponding operational semantics law. The operational semantics of SFP does not reduce an expression containing  $\top$  to  $\top$  unless the outermost expression is a sequence and has  $\top$  as one of its entries. Therefore it would not be generally appropriate to replace  $[..., \bar{\top}, ...]$  with  $\bar{\top}$ . Note, too, that such a law would be incompatible with law I.6, for then we could replace  $1 \circ [f_1, \bar{\top}]$  with either  $f_1$  or  $\bar{\top}$ .

Law III.3 (pair  $\rightarrow 1 \circ distr \equiv [1 \circ 1, 2]$ ) is the only law that does not extend gracefully to the SFP domain. The law fails for objects that are pairs whose first



entry is  $\perp$  or  $\langle \cdot \rangle$  for either semantics. The part of the domain for which the law does not hold is very small, so not much has been affected, but the increased complexity of the hypothesis makes it more difficult to understand.

The gap in this law uncovers a property of the functional forms with respect to the extension mechanism for primitive functions given in Chapter 3. Some of the functional forms, such as construction, preserve the extension of functions, while others, such as composition, do not. That is, for all monotonic functions  $f$  and  $g$  on  $\mathbf{B}$ ,  $([f g])' = [f' g']$  where for any monotonic function  $h$  on  $\mathbf{B}$ ,  $h'$  is the extension of  $h$  to  $\mathbf{C}$  as given in Definition 39 in Chapter 3. However, for some monotonic functions  $f$  and  $g$  on  $\mathbf{B}$ , it is not true that  $(f \circ g)' = f' \circ g'$ . For example,  $(1 \circ \text{distr})' \neq 1' \circ \text{distr}'$ , since they do not agree on  $\langle \langle \cdot \rangle, x \rangle$ :

$$\begin{aligned}
(1 \circ \text{distr})'(\langle \langle \cdot \rangle, x \rangle) &= \text{glb}\{(1 \circ \text{distr})(y) \mid \langle \langle \cdot \rangle, x \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= \text{glb}\{(1 \circ \text{distr})(y) \mid y = \langle \langle \bigwedge_{i=1}^n w_i, z \rangle, 0 \leq n < \infty, \\
&\quad w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z \} \\
&= \text{glb}\{1(v) \mid v = \text{distr}(\langle \langle \cdot \rangle, z \rangle), x \sqsubseteq z \text{ or} \\
&\quad v = \text{distr}(\langle \langle \bigwedge_{i=1}^n w_i, z \rangle), n \geq 1, w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z \} \\
&= \text{glb}\{1(v) \mid v = \top \text{ or} \\
&\quad v = \langle \bigwedge_{i=1}^n \langle w_i, z \rangle, n \geq 1, w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z \} \\
&= \text{glb}\{u \mid u = 1(\top) \text{ or} \\
&\quad u = 1(\langle \bigwedge_{i=1}^n \langle w_i, z \rangle), n \geq 1, w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z \} \\
&= \text{glb}\{t \mid t = \top \text{ or} \\
&\quad t = \langle w_1, z \rangle, w_1, z \in \mathbf{B} \ \& \ x \sqsubseteq z \} \\
&= \langle \perp, x \rangle
\end{aligned}$$

$$\begin{aligned}
(1' \circ \text{distr}')(\langle \perp, x \rangle) &= 1' : \text{glb}\{\text{distr}(y) \mid \langle \perp, x \rangle \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= 1' : \text{glb}\{\text{distr}(y) \mid y = \langle \bigwedge_{i=1}^n w_i, z \rangle, \ 0 \leq n < \infty, \\
&\quad w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z\} \\
&= 1' : \text{glb}\{v \mid v = \top \text{ or} \\
&\quad v = \langle \bigwedge_{i=1}^n \langle w_i, z \rangle, \ n \geq 1, \ w_i, z \in \mathbf{B} \ \& \ x \sqsubseteq z\} \\
&= 1'(\langle \bigwedge_{i=1}^n \langle \perp, x \rangle, \ \text{where } n \geq 1) \\
&= \text{glb}\{1(y) \mid \langle \bigwedge_{i=1}^n \langle \perp, x \rangle \sqsubseteq y, \ n \geq 1, \ \& \ y \in \mathbf{B}\} \\
&= \text{glb}\{1(y) \mid y = \langle \bigwedge_{i=1}^n \langle w, z \rangle, \ n \geq 1, \ w, z \in \mathbf{B} \ \& \ x \sqsubseteq z\} \\
&= \text{glb}\{u \mid u = w \ \& \ w \in \mathbf{B}\} \\
&= \perp
\end{aligned}$$

However, it is true that  $f' \circ g' \leq (f \circ g)'$ .

**Theorem 55:** For all monotonic functions  $f, g$  on  $\mathbf{B}$ ,  $([f \ g])' = [f' \ g']$ , and  $f' \circ g' \leq (f \circ g)'$ .

**Proof:** To show:  $([f \ g])' = [f' \ g']$ .

For  $x \in \mathbf{C}$ ,

$$\begin{aligned}
[f' \ g'](x) &= \langle f'(x) \ g'(x) \rangle \\
&= \langle \text{glb}\{f(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \ \text{glb}\{g(y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \rangle \\
&= \text{glb}\{\langle f(y) \ g(y) \rangle \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= \text{glb}\{[f \ g](y) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= ([f \ g])'(x).
\end{aligned}$$

To show:  $f' \circ g' \leq (f \circ g)'$ .

For  $x \in \mathbf{C}$ ,

$$\begin{aligned}
(f' \circ g')(x) &= f'(g'(x)) \\
&= \text{glb}\{f(z) \mid g'(x) \sqsubseteq z \ \& \ z \in \mathbf{B}\} \\
&\leq \text{glb}\{f(g(y)) \mid g'(x) \sqsubseteq g(y), \ x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= \text{glb}\{f(g(y)) \mid x \sqsubseteq y \ \& \ y \in \mathbf{B}\} \\
&= (f \circ g)'(x).
\end{aligned}$$

Here we note that for all  $y \in \mathbf{B}$  such that  $x \sqsubseteq y$ ,  $g'(x) \sqsubseteq g(y)$ , though there may be other elements that satisfy the conditions on  $z$ .

■

We would like to have that, for any functional form  $F$  with operands  $f_1, \dots, f_n$ ,  $(F(f_1, \dots, f_n))' = F(f'_1, \dots, f'_n)$ , for then it would be simple to extend all the functions on  $\mathbf{B}$  to functions on  $\mathbf{C}$  by Definition 39. But since it is the case that some functional forms, such as composition, do not obey this law, we must content ourselves with extending the primitive functions by Definition 39, writing new definitions for the functional forms, and describing a set of functions on  $\mathbf{C}$  from the extended primitive functions and the modified functional forms.

One law for FP not mentioned in the Turing Award Lecture is  $trans \circ trans \leq id$ . The corresponding SFP law (for either semantics) is  $trans \circ trans \geq id$ . The reversal of the inequality is not quite as simple as with the previously discussed laws. It is true that when transpose fails on an operand, the SFP result is  $\top$ . But there is more to the inequality than just those cases for which transpose produces  $\top$ . Consider the element  $\langle \langle 1 \dagger, \langle 2, 4 \dagger \rangle \rangle$ . Applying  $trans$  to this element produces  $\langle \langle 1, 2 \dagger, \langle \perp, 4 \dagger \rangle \rangle$ , and applying  $trans$  to that result produces  $\langle \langle 1, \perp \dagger, \langle 2, 4 \dagger \rangle \rangle$ , an element greater than the original argument. This is a consequence of the extension mechanism. The extension mechanism attempts to anticipate what will happen next in the development of an argument by considering all of the complete objects that are suitable operands to the primitive function, which the approximation can grow in to. In the case above, for the argument to  $trans$  to be suitable, all entry sequences must be of the same length. Thus the extension supplies  $\perp$  where arguments are expected to develop if all goes well.

Another FP law not mentioned in the Turing Award Lecture is  $rev \circ rev \leq id$ . This particular law has no corresponding SFP law. If reverse is applied to an inappropriate argument, such as an atom, then it produces  $\top$ , just as the FP version produces  $\perp$ . We might, therefore, expect the law  $rev \circ rev \geq id$  to hold, but it does

not. The peculiar behavior noted with transpose appears in another form here. If reverse is applied to a prefix, such as  $\langle 1, 2 \rangle$ , then the result is  $\langle \perp, \perp \rangle$ , since we cannot know what the first and second entries of the result will be, only that there are at least two of them. Hence  $rev \circ rev : \langle 1, 2 \rangle = \langle \perp, \perp \rangle \leq id : \langle 1, 2 \rangle$ . The fact that this FP law has no corresponding SFP law is not particularly disturbing, since SFP has streams, and it is not meaningful to reverse a stream or even a prefix, since a prefix approximates a stream.

The law  $trans \circ trans \geq id$  illustrates a property of the primitive function *trans*. There are many such laws, that is, laws involving properties of particular primitive functions. Furthermore, defining new functions creates new laws. Backus did not include such laws in his list, presumably because he was interested primarily in the properties of functional forms. The list below contains a few of these laws that will be used in the last example of Chapter 6.

**Def**  $union := apndl \circ [1 \circ 1, apndl \circ [2 \circ 1, union \circ tail]]$

**Def**  $group2 := apndl \circ [[1 \ 2], group2 \circ tail \circ tail]$

*union* takes a stream of pairs and unpairs them, that is, it removes one level of sequence brackets from each item in the stream. *group2* changes a stream into a stream of pairs.

## V Primitive and defined functions

V.1  $trans \circ trans \geq id$

V.2  $trans \circ apndl \circ [apndl \circ [f_1, f_2], apndl \circ [g_1, g_2]] \equiv apndl [[f_1, g_1], trans [f_2, g_2]]$

V.3  $union \circ group2 \equiv id$  for streams

## Conclusion

The algebra that Backus gave for FP carries over to SFP with very few changes. Most of the changes arise from the fact that  $\perp$  represents error in FP, whereas  $\top$  is the error object in SFP. The laws for SFP also depend on whether the denotational semantics or the operational semantics is used. All of these differences occur because the operational semantics uses an outermost evaluation rule, defining a slightly different language from that given by the denotational semantics.

## Chapter 6

### Examples

SFP is an extension of FP that allows meaningful non-terminating computations with arbitrarily good approximations to infinite objects. We have chosen to extend FP because of its useful mathematical properties. In this chapter we exhibit several examples that demonstrate the use of the language for non-terminating computation and how the algebra can be used to prove properties of programs.

The reader who has programmed in FP will immediately notice how close the SFP programs are to FP programs. The main difference is the lack of conditionals in recursively defined SFP programs.

At the time of this writing, there is no implementation for SFP. The program examples given here have not been written with efficiency as a major consideration, since more efficient versions of these programs may be too obscure for the purposes of illustration.

**Example 1:** Many problems require the generation of a simple stream from which the output will be computed. Frequently, the desired stream is an arithmetic progression, such as a stream whose only value is 1, or the stream of integers. (A geometric progression can be produced by a straightforward modification of the program given for arithmetic progressions.) Given below is a program *arith.prog* to generate an arithmetic progression:

$$\langle x, \langle x + k, \langle x + 2 * k, \dots \langle x + n * k, \dots \rangle \dots \rangle \rangle \rangle$$

from an input argument of the form  $\langle \text{initial\_value}, \text{increment} \rangle = \langle x, k \rangle$ . Note that the output is a pair, the second element of which is infinitely nested.

$$\text{Def arith.prog} := [ 1, \text{arith.prog} \circ [+ , 2] ]$$

**Example 2:** Streams are often defined to be pairs of the form  $\langle first, rest \rangle$ , where *first* is the head of the stream and *rest* is a sequence that is the tail. (Example 1 illustrated this definition of stream.) There is an obvious correspondence between such a ‘tree’ representation of a stream and the ‘flat’, or sequence, representation. Either representation, however, can be used in the domain **D**. The following programs convert streams of the form  $\langle first, rest \rangle$  to the ‘flat’ form and back again:

**Def** flat := *apndl* ◦ [ 1, flat ◦ 2 ] (tree to flat)

**Def** tree := [ 1, tree ◦ tail ] (flat to tree)

**Example 3:** *The Sieve of Eratosthenes.* This example uses the function *arith.prog* defined in Example 1 and the *flat* function defined in Example 2. It also uses two primitive functions not given in the Turing Award Lecture [Backus 1978]. The function *eq0* returns  $\perp$  if the argument is  $\perp$ ,  $\top$  if the argument is  $\top$ , ‘true’ if the argument is the number zero, and ‘false’ if it is anything else. The function *mod* expects a pair of numbers as its argument and returns the remainder of the integer division of the first number by the second number. The input to the program *primes* should be the pair  $\langle 2, 1 \rangle$ , which will cause *arith.prog* to produce the nested stream  $\langle 2, \langle 3, \dots \rangle \rangle$ .

**Def** primes := sieve ◦ flat ◦ *arith.prog*

*sieve* is the function that actually produces the primes from its input, and *flat* is used to flatten the nested stream (tree) of primes to a non-nested stream.

**Def** sieve := [ 1, sieve ◦ filter ]

The input to *sieve* is a nested stream of natural numbers, the first of which is the next prime and the tail of which is to be filtered to remove multiples of that prime. Each time *sieve* is invoked via a recursive call, a new prime is generated as output and a new filter is established.

**Def** filter := *eq0* ◦ *mod* ◦ [ 1 ◦ 2, 1 ] → filter ◦ [ 1, 2 ◦ 2 ]; [ 1 ◦ 2, filter ◦ [ 1, 2 ◦ 2 ] ]

The input to *filter* is a nested stream of integers. *filter* removes from this stream all multiples of the first element and returns the filtered stream as output. The program *filter* is equal to  $p \rightarrow f;g$ , where  $p$  is a predicate that tests whether the second item in the stream is multiple of the first item. If it is a multiple, then  $f$  removes the second item and filters the rest of the stream for the first item. If the second item is not a multiple, then  $g$  removes the second item from the input, puts it as the first element of the output stream, and then filters the rest of the stream for the first item. The expression  $[1, 2 \circ 2]$  takes a stream and removes the second item, leaving the first and other items in place.

**Example 4:** A ‘history-sensitive’ stream function. This approach is being developed in order to embed schemes similar to Backus’s AST system in this language framework. We describe a function that accepts an input of the form  $\langle s_1, x_1, x_2, x_3, \dots \rangle$ , where  $s_1$  is an initial system state and the sequence  $x_1, x_2, x_3, \dots$  is a sequence of inputs to the system, and whose output is a sequence  $\langle o_1, o_2, o_3, \dots \rangle$ . Mathematically the function will map one infinite object to another, but it can be viewed as a system with functions that, for each  $i$ , use the current state  $s_i$  and the current input  $x_i$  to compute the next output  $o_i$  and the next state  $s_{i+1}$ .

A simple example of a program that might be written with this function is a screen editor. The value of the edited file at any point would be the state at that point. The sequence of inputs  $x_1, x_2, x_3, \dots$  would be a stream of commands to edit or move about in the file. The output at each point would be the part of the updated file to be written to the screen.

We define a functional form, denoted by braces, that requires two function arguments, a ‘next state’ function  $S^*$  and an ‘output’ function  $O^*$ . The function  $S^*$  maps the domain  $\mathbf{S} \times \mathbf{O}$  (where  $\mathbf{S}$  is the set of states and  $\mathbf{O}$  is the set of objects) into  $\mathbf{S}$ , and  $O^*$  maps the same domain into  $\mathbf{O}$ . (The set  $\mathbf{S}$  can be chosen to be any subset of the set of objects  $\mathbf{O}$ .) The effect of applying a function constructed with this functional form applied to a stream argument can be described as follows:

$$(\{O^*, S^*\} : \langle s_1, x_1, x_2, x_3, \dots \rangle) =$$

$$(apndl := (O* := \langle s_1, x_1 \rangle), (\{O*, S*\} \circ apndl := \langle S* := \langle s_1, x_1 \rangle, \langle x_2, x_3, \dots \rangle \rangle))$$

Thus, the value of  $S* := \langle s_i, x_i \rangle$  is the next state  $s_{i+1}$  and the value of  $O* := \langle s_i, x_i \rangle$  is  $o_i$ , the result of processing input  $x_i$  when in state  $s_i$ . The definition of the functional form denoted by  $\{\}$  can be given entirely within FP:

$$\{f, g\} := apndl \circ [ f \circ [ 1, 2 ], \{f, g\} \circ apndl \circ [ g \circ [ 1, 2 ], tail \circ tail ] ]$$

Here  $f$  computes the next output and  $g$  computes the next state.

There are other ways of describing similar functions based on extensions of FP functions; in particular, a functional form can be described such that the system state is an object parameter of the functional form rather than the first entry of a stream input. The function described above produces one entry  $o_i$  for each input entry  $x_i$ , but other functions can be defined that produce different or varying numbers of output entries  $o_i, o_{i+1}, \dots, o_{i+k}$  for a single input entry  $x_i$ .

**Example 5:** A program to generate the cartesian product of a pair of streams.

$$\mathbf{Def} \text{ cp} := \text{zip} \circ [ f, g ]$$

The input to  $cp$  is a pair of streams. The output is the cartesian product of that pair.

$$\mathbf{Def} f := \text{distl} \circ [ 1 \circ 1, 2 ]$$

The input to  $f$  is a pair of streams. The output is a stream of pairs.  $f$  takes the first element of the first stream and pairs it with every element of the second stream.

$$\mathbf{Def} g := \text{cp} \circ [ \text{tail} \circ 1, 2 ]$$

The input to  $g$  is a pair of streams. The output is a stream of pairs.  $g$  generates the cartesian product of two streams, the first being the tail of the first stream in the input and the second being the second stream in the input.

$$\mathbf{Def} \text{ zip} := apndl \circ [ 1 \circ 1, apndl \circ [ 1 \circ 2, \text{zip} \circ \text{ctail} ] ]$$



If the input to *zip* is a pair of streams of pairs then the output is a stream of pairs. *zip* creates a stream of pairs by alternately choosing the next pair from the two input streams.

**Example 6:** The algebra can be used to show the equivalence of two programs. For example, the program  $flat \circ arith.prog$  from Examples 1 and 2 first produces an arithmetic progression in tree form and then flattens it into a stream form. This can be shown to be equivalent to the following program:

**Def**  $a.p2 := apndl \circ [1, a.p2 \circ [+ , 2]]$

Proof:

By definition of *arith.prog*,

$$flat \circ arith.prog = flat \circ [1, arith.prog \circ [+ , 2]]$$

(by definition of *flat*)

$$= apndl \circ [1, flat \circ 2] \circ [1, arith.prog \circ [+ , 2]]$$

(by law I.1)

$$= apndl \circ [1 \circ [1, arith.prog \circ [+ , 2]], flat \circ 2 \circ [1, arith.prog \circ [+ , 2]]]$$

(by law I.5)

$$= apndl \circ [1, flat \circ arith.prog \circ [+ , 2]]$$

**Example 7:** This example traces a few steps in the execution of the program given in Example 6. Applying *a.p2* to the initial input  $\langle 1, 2 \rangle$  should produce the odd positive integers as a stream. We show a few steps in the reduction of  $(a.p2 : \langle 1, 2 \rangle)$  to illustrate the re-writing rules:

(Parentheses, indicating application, are omitted.)

$$\begin{aligned} a.p2 : \langle 1, 2 \rangle &\rightarrow apndl \circ [1, a.p2 \circ [+ , 2]] : \langle 1, 2 \rangle \\ &\rightarrow apndl : [1, a.p2 \circ [+ , 2]] : \langle 1, 2 \rangle \\ &\rightarrow apndl : \langle 1 : \langle 1, 2 \rangle , a.p2 \circ [+ , 2] : \langle 1, 2 \rangle \rangle \\ &\rightarrow \langle 1 : \langle 1, 2 \rangle \sim a.p2 \circ [+ , 2] : \langle 1, 2 \rangle \rangle \\ &\rightarrow \langle 1 \sim a.p2 : [+ , 2] : \langle 1, 2 \rangle \rangle \end{aligned}$$

$\rightarrow \langle 1 \sim \text{apndl} \circ [1, \text{a.p2} \circ [+ , 2]] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
 $\rightarrow \langle 1 \sim \text{apndl} : [1, \text{a.p2} \circ [+ , 2]] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
 $\rightarrow \langle 1 \sim \text{apndl} : \langle 1 : [+ , 2] : \langle 1, 2 \rangle , \text{a.p2} \circ [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle \rangle$   
 $\rightarrow \langle 1 \sim \langle 1 : [+ , 2] : \langle 1, 2 \rangle \sim \text{a.p2} \circ [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle \rangle$   
 $\rightarrow \langle 1, 1 : [+ , 2] : \langle 1, 2 \rangle \sim \text{a.p2} \circ [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
 $\rightarrow \langle 1, 1 : \langle + : \langle 1, 2 \rangle , 2 : \langle 1, 2 \rangle \rangle \sim \text{a.p2} : [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
 $\rightarrow \langle 1, + : \langle 1, 2 \rangle \sim \text{apndl} \circ [1, \text{a.p2} \circ [+ , 2]] : [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
 $\rightarrow \langle 1, 3 \sim \text{apndl} : [1, \text{a.p2} \circ [+ , 2]] : [+ , 2] : [+ , 2] : \langle 1, 2 \rangle \rangle$   
*etc.*

**Example 8:** Finally, we give an example of a proof of a property of a program. The example is due to Kahn [Kahn 1974]. Kahn defines four functions operating on and producing streams, interconnected as shown by the directed graph of Figure 1. Names are assigned to the I/O interconnection arcs, which he calls channels. The function  $f$  has two inputs,  $Y$  and  $Z$ , which  $f$  merges into an output stream,  $X$ . The function  $g$  takes the stream  $X$  as input and separates it into two output streams  $T1$  and  $T2$  by alternately directing the next element of the stream to first  $T1$  and then  $T2$ . The function  $h$  has two instantiations, which Kahn labels  $h_0$  and  $h_1$ . The function  $h$  sends out an initial value, which is 0 for  $h_0$  and 1 for  $h_1$ , then receives its input,  $T1$  for  $h_0$  and  $T2$  for  $h_1$ , and passes it through unchanged as output.  $Y$  is the output of  $h_0$  and  $Z$  is the output of  $h_1$ . Kahn then shows (using structural induction) that the history of  $X$  is an alternating stream of 0s and 1s.

For the same program graph, SFP programs can be defined with the same functions as the corresponding Kahn functions. Each arc will be realized as an application of one of the SFP programs to the input(s) of that program. We define the functions  $f$ ,  $g$ , and  $h$ , describe the arcs  $X$ ,  $Y$ ,  $Z$ ,  $T1$ , and  $T2$  and then show that  $X = \langle 0, 1 \sim X \rangle$ .

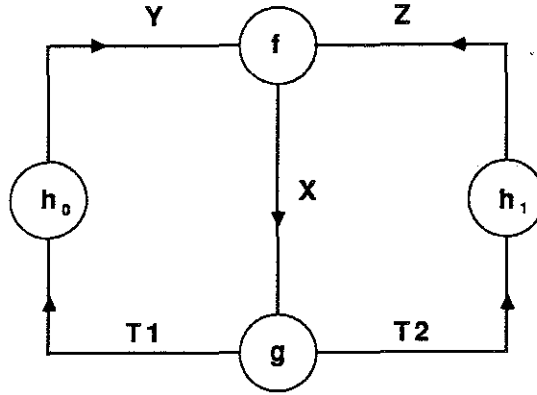


Figure 1: Kahn's Example.

Definition of the functions:

**Def**  $f := \text{union} \circ \text{trans}$

**Def**  $\text{union} := \text{apndl} \circ [1 \circ 1, \text{apndl} \circ [2 \circ 1, \text{union} \circ \text{tail}]]$

**Def**  $\text{group2} := \text{apndl} \circ [[1 \ 2], \text{group2} \circ \text{tail} \circ \text{tail}]$

**Def**  $g := \text{trans} \circ \text{group2}$

**Def**  $h_0 := \text{apndl} \circ [\bar{0}, \text{id}]$  and  $h_1 := \text{apndl} \circ [\bar{1}, \text{id}]$

$\text{union}$  takes a stream of pairs and unpairs them, that is, it removes one level of sequence brackets from each item in the stream.  $\text{group2}$  changes a stream into a stream of pairs.  $g$  changes a stream into a pair of streams.  $h_0$  appends a 0 to the beginning of its input stream;  $h_1$  appends a 1 to the beginning of its input stream.

Definition of the channels:

$Y = h_0 : T1$

$Z = h_1 : T2$

$X = f : \langle Y, Z \rangle$

$\langle T1, T2 \rangle = g : X$

Proof that  $X = \langle 0, 1 \sim X \rangle$ :

$$\begin{aligned}
X &= f : \langle Y, Z \rangle \\
&\text{(by definition of } f\text{)} \\
&= \text{union} \circ \text{trans} : \langle Y, Z \rangle \\
&\text{(by definition of } Y \text{ and } Z\text{)} \\
&= \text{union} \circ \text{trans} : \langle h_0 : T1, h_1 : T2 \rangle \\
&\text{(by substitution of equivalent expressions)} \\
&= \text{union} \circ \text{trans} \circ [h_0 \circ 1, h_1 \circ 2] : \langle T1, T2 \rangle \\
&\text{(by definition of } h_0 \text{ and } h_1\text{)} \\
&= \text{union} \circ \text{trans} \circ [\text{apndl} \circ [\bar{0}, \text{id}] \circ 1, \text{apndl} \circ [\bar{1}, \text{id}] \circ 2] : \langle T1, T2 \rangle \\
&\text{(by laws I.1, III.1, and III.2)} \\
&= \text{union} \circ \text{trans} \circ [\text{apndl} \circ [\bar{0}, 1], \text{apndl} \circ [\bar{1}, 2]] : \langle T1, T2 \rangle \\
&\text{(by law V.1)} \\
&= \text{union} \circ \text{apndl} \circ [[\bar{0}, \bar{1}], \text{trans} \circ [1, 2]] : \langle T1, T2 \rangle \\
&\text{(by evaluation)} \\
&= \text{union} \circ \text{apndl} : \langle \langle 0, 1 \rangle, \text{trans} : \langle T1, T2 \rangle \rangle \\
&\text{(by definition of the pair of streams } \langle T1, T2 \rangle\text{)} \\
&= \text{union} \circ \text{apndl} : \langle \langle 0, 1 \rangle, \text{trans} \circ g : X \rangle \\
&\text{(by definition of } g\text{)} \\
&= \text{union} \circ \text{apndl} : \langle \langle 0, 1 \rangle, \text{trans} \circ \text{trans} \circ \text{group2} : X \rangle \\
&\text{(by the algebraic law that } \text{trans} \circ \text{trans} \geq \text{id}\text{)} \\
&\geq \text{union} \circ \text{apndl} : \langle \langle 0, 1 \rangle, \text{group2} : X \rangle \\
&\text{(by definition of } \text{union}\text{)} \\
&= \text{apndl} \circ [1 \circ 1, \text{apndl} \circ [2 \circ 1, \text{union} \circ \text{tail}]] \circ \text{apndl} : \langle \langle 0, 1 \rangle, \text{group2} : X \rangle \\
&\text{(by evaluation)} \\
&= \text{apndl} : \langle 0, \text{apndl} : \langle 1, \text{union} \circ \text{group2} : X \rangle \rangle \\
&\text{(because } \text{union} \circ \text{group2} = \text{id}\text{)} \\
&= \text{apndl} : \langle 0, \text{apndl} : \langle 1, X \rangle \rangle \\
&\text{(by evaluation)} \\
&= \langle 0, 1, \sim X \rangle
\end{aligned}$$

## *Chapter 7*

### *Conclusions and Future Work*

The computer industry is facing a software crisis. Code that has been and is being written is often unreliable, difficult to maintain, and impossible to verify. The past two decades have produced three generations of hardware. But languages have lagged behind in spite of the recognized deficiencies of those currently used. A new generation of languages is long overdue.

Several properties appear to be attractive in the design of new languages. More powerful features, such as higher order functions and more powerful primitive commands would free the programmer from some of the tedious detail with which he must currently cope. Another desirable aspect is found in languages that facilitate proofs of properties of programs or even of the languages themselves. The ability to do automatic optimization needs to be improved so that it can be done at a higher level than is currently feasible.

It is difficult to imagine modifying procedural languages, such as ALGOL, to incorporate gracefully these new properties. A new breed of languages is required. The ultimate goal is a new kind of programming environment. This environment will certainly encompass new languages, more powerful tools, and perhaps hardware designed specifically for the new languages. One of the best hopes lies with functional and logic languages. These languages would provide a useful tool. In order to compete with existing languages, they need an efficient implementation, which may require radical changes to computer architecture.

FP is a functional language that meets many of the criteria described above for language design extremely well, but it has a number of deficiencies that prevent

it from being used to develop a new programming environment. One of these deficiencies is the inability to write (useful) non-terminating programs.

The research presented here has demonstrated that FP can be extended to include infinite objects in such a way that the mathematical properties of the language are preserved. The new language, SFP, has a syntax nearly identical to that of FP. The FP language is a subset of SFP, and most FP expressions have the same values as the corresponding SFP expressions. The algebra has been carried over with only minor changes. Furthermore, we have shown that any SFP program is monotonic and continuous, which makes it an acceptable computing language.

The extension mechanism given in Chapter 3 greatly facilitated the proof that all programs in SFP are continuous in that it provided a uniform way to extend any monotonic primitive function, and thus the proof of continuity was general rather than specific to each primitive function. Though the extension mechanism from  $\mathbf{C}$  to  $\mathbf{D}$  is a standard construction, the mechanism for extending functions from  $\mathbf{B}$  to  $\mathbf{C}$  is new and is a substantial result of this research. In addition to facilitating the proof of continuity, the extension mechanism has another important property: it produces maximal monotonic extensions.

Further improvements to the SFP language are needed for it to provide a basis for a programming environment with useful tools.

An obvious deficiency is the lack of higher-order functions. This might be approached by extending SFP to something like FFP, a formal version of FP with higher-order functions. An alternative approach would be to extend FP 1.5 [Arvind *et al.* 1982] to streams using mechanisms similar to those given here.

Another need of the SFP language is some way to accommodate non-determinism. One property of streams in actual practice is that the elements are generally produced sequentially, and an arbitrary amount of time may elapse between two successive elements. Consider the *zip* function of Example 5 of the previous chapter. The input to *zip* consists of two streams, and the output is a merge

of the two input streams so that the next output item is chosen alternately from the two input streams. In many cases, it may not be important that the output alternate. If so, it is not necessary to wait on the next item of one stream if items are already available in the other stream. The SFP language does not permit the programmer to specify a non-deterministic choice between two things. A non-deterministic merge would be more useful than *zip* in many cases, and it warrants investigation.

A substantial problem with the operational semantics we describe occurs with SFP functions that map (potentially) infinite inputs to (potentially) infinite outputs. One envisions that potentially infinite inputs may be acquired incrementally over a period of time. In this case, it is desirable to begin producing output before all of the input has arrived. Using the operational semantics given here, any improvement to the input would require that a new approximation to the output be computed entirely anew; all parts of the previous approximation are recomputed. This method, though quite straightforward, is unacceptably inefficient. What is needed is an implementation that is capable of modifying the output based on new information about the input without recomputing what has already been computed.

An infinite set in SFP can be specified algorithmically. It is sometimes more convenient to specify a set by the properties of its elements, as is done in Zermelo-Frankel set abstraction, rather than by specifying an algorithm to generate the set. The KRC language [Turner 1982] uses this specification method, and it would be an attractive feature to add to SFP.

One of the problems of the FP language is that programmers find it difficult to understand the FP programs. Because the syntax of SFP is nearly identical, the same problem occurs with SFP programs. Part of the solution may be better programming habits, but improvements can also be made to make the syntax more palatable. An investigation of solutions to the problem can start by integrating Backus's extended definitions [Backus 1981] with streams.

The use of  $\top$  as the sole error element is mathematically elegant but somewhat restrictive. Another approach to error representation that has been considered is to have a number of error objects, one at the end of each "branch" in the lattice. Though some research has been done in this area, it is not yet clear whether or not such an approach can be made to work cleanly.

The most obvious requirement for a real environment containing SFP is an implementation of the language. Since SFP is a parallel language, a parallel processor would take best advantage of the language's parallelism. The UNC tree machine, a fine grain multi-processor designed by Magó [Magó 1979, 1980, 1984], is a hardware interpreter for the FFP language. The similarities of SFP to FFP suggest that it would be worthwhile to investigate modifying the tree machine to execute SFP. Direct hardware implementations of SFP appear to be some years into the future, and since the issues involve much more than parallelism, it is also worthwhile to implement SFP in software on a conventional machine. Such an implementation would provide an opportunity to investigate the feasibility of other features of the language, such as automatic program transformation.

Along with an implementation of the language, tools, such as editors and debuggers, are needed to create a real programming environment. Adding types to the language or having an automatic type inference system available would assist the SFP programmer in keeping track of his data structures. Another tool to investigate is a translator from another language, such as KRC, to SFP (with higher order functions). This would allow the programmer to program in a different language but would still give him access to the powerful SFP algebra, which could be used to transform or verify the program.

Finally, the algebra needs to be developed. New laws need to be added, and software making use of the laws needs to be written. The algebra could be used for a number of applications, including automatic program transformation for optimization, proving properties of programs, or analyzing time and space requirements.



## Appendix A

### The FP Primitive Functions and Functional Form Operators

This appendix contains the definitions of the primitive functions and functional forms as given by Backus in his Turing Award Lecture.

#### Selector

$$1 : x \equiv x = \langle x_1 \dots x_n \rangle \rightarrow x_1;$$

$\perp$

and for any positive integer  $s$

$$s : x \equiv x = \langle x_1 \dots x_n \rangle \ \& \ n \geq s \rightarrow x_s;$$

$\perp$

#### Tail

$$tl : x \equiv x = \langle x_1 \rangle \rightarrow \phi;$$

$$x = \langle x_1 \dots x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_2 \dots x_n \rangle;$$

$\perp$

#### Identity

$$id : x \equiv x$$

#### Atom

$$atom : x \equiv x \text{ is an atom} \rightarrow T;$$

$$x \neq \perp \rightarrow F;$$

$\perp$

## Equals

$$eq : x \equiv x = \langle y, z \rangle \ \& \ y = z \rightarrow T;$$

$$x = \langle y, z \rangle \ \& \ y \neq z \rightarrow F;$$

⊥

## Null

$$null : x \equiv x = \phi \rightarrow T;$$

$$x \neq \perp \rightarrow F;$$

⊥

## Reverse

$$reverse : x \equiv x = \phi \rightarrow \phi;$$

$$x = \langle x_1 \dots x_n \rangle \rightarrow \langle x_n \dots x_1 \rangle;$$

⊥

## Distribute from left; distribute from right

$$distl : x \equiv x = \langle y, \phi \rangle \rightarrow \phi;$$

$$x = \langle y, \langle z_1 \dots z_n \rangle \rangle \rightarrow \langle \langle y, z_1 \rangle \dots \langle y, z_n \rangle \rangle;$$

⊥

$$distr : x \equiv x = \langle \phi, y \rangle \rightarrow \phi;$$

$$x = \langle \langle y_1 \dots y_n \rangle, z \rangle \rightarrow \langle \langle y_1, z \rangle \dots \langle y_n, z \rangle \rangle;$$

⊥

## Length

$$length : x \equiv x = \langle x_1 \dots x_n \rangle \rightarrow \mathbf{n};$$

$$x = \phi \rightarrow 0;$$

⊥

## Add, subtract, multiply and divide

$$+ : x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y + z;$$

⊥

$$- : x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y - z;$$

⊥

$$\times : x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y \times z;$$

⊥

$$\div : x \equiv x = \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y \div z;$$

⊥

### Transpose

$$trans : x \equiv x = \langle \phi, \dots, \phi \rangle \rightarrow \phi;$$

$$x = \langle x_1 \dots x_n \rangle \rightarrow \langle y_1 \dots y_m \rangle;$$

⊥

where  $x_i = \langle x_{i1} \dots x_{im} \rangle$  &  $y_j = \langle x_{1j} \dots x_{nj} \rangle$ , for  $1 \leq i \leq n$ ,  $1 \leq j \leq m$

### And, or, not

$$and : x \equiv x = \langle T, T \rangle \rightarrow T;$$

$$x = \langle T, F \rangle \vee \langle F, T \rangle \vee \langle F, F \rangle \rightarrow F;$$

⊥

etc.

### Append left; append right

$$apndl : x \equiv x = \langle y, \phi \rangle \rightarrow \langle y \rangle;$$

$$x = \langle y, \langle z_1 \dots z_n \rangle \rangle \rightarrow \langle y, z_1 \dots z_n \rangle;$$

⊥

$$apndr : x \equiv x = \langle \phi, z \rangle \rightarrow \langle z \rangle;$$

$$x = \langle \langle y_1 \dots y_n \rangle, z \rangle \rightarrow \langle y_1 \dots y_n, z \rangle;$$

⊥

### Right selectors; right tail

$$1r : x \equiv x = \langle x_1 \dots x_n \rangle \rightarrow x_n;$$

⊥

$$2r : x \equiv x = \langle x_1 \dots x_n \rangle \ \& \ n \geq 2 \rightarrow x_{n-1};$$

⊥

etc.

$$\begin{aligned}
\text{tlr} : x &\equiv x = \langle x_1 \rangle \rightarrow \phi; \\
&x = \langle x_1 \dots x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_1 \dots x_{n-1} \rangle; \\
&\perp
\end{aligned}$$

### Rotate left; rotate right

$$\begin{aligned}
\text{rotl} : x &\equiv x = \phi \rightarrow \phi; \\
&x = \langle x_1 \rangle \rightarrow \langle x_1 \rangle; \\
&x = \langle x_1 \dots x_n \rangle \ \& \ n \geq 2 \rightarrow \langle x_2 \dots x_n, x_1 \rangle; \\
&\perp
\end{aligned}$$

etc.

### Composition

$$(f \circ g)(x) \equiv f(g(x))$$

### Construction

$$[f_1, \dots, f_n](x) \equiv \langle f_1(x), \dots, f_n(x) \rangle$$

### Condition

$$\begin{aligned}
(p \rightarrow f; g)(x) &\equiv p(x) = T \rightarrow f(x); \\
&p(x) = F \rightarrow g(x); \\
&\perp
\end{aligned}$$

### Constant

$$\begin{aligned}
\bar{x}(y) &\equiv y = \perp \rightarrow \perp; \\
&x
\end{aligned}$$

### Insert

$$\begin{aligned}
/f(x) &\equiv x = \langle x_1 \rangle \rightarrow x_1; \\
&x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f(\langle x_1, /f(\langle x_2, \dots, x_n \rangle) \rangle); \\
&\perp
\end{aligned}$$

### Apply to all

$$\begin{aligned}
\alpha f(x) &\equiv x = \phi \rightarrow \phi; \\
&x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle; \\
&\perp
\end{aligned}$$

## Binary to unary

$$(bu\ f\ x)(y) \equiv f(\langle x, y \rangle)$$

## While

$$(while\ p\ f)(x) \equiv p(x) = T \rightarrow (while\ p\ f)(f(x));$$

$$p(x) = F \rightarrow x;$$

⊥

*Appendix B*  
*The SFP Primitive Functions and Functional Form Operators*  
*Denotational Semantics*

This appendix contains a list of the SFP primitive functions and functional forms as defined by the denotational semantics of SFP.

**Selector**

$$\begin{aligned}
 l(x) \equiv & \quad x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n < \infty \rightarrow x_1; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n \leq \infty \rightarrow x_1; \\
 & \quad \top
 \end{aligned}$$

and for any positive integer  $s$

$$\begin{aligned}
 s(x) \equiv & \quad x = \perp \rightarrow \perp; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ s \leq n < \infty \rightarrow x_s; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < s \rightarrow \perp; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ s \leq n \leq \infty \rightarrow x_s; \\
 & \quad \top
 \end{aligned}$$

**Tail**

$$\begin{aligned}
 tl(x) \equiv & \quad x = \perp \rightarrow \perp; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \langle \rangle_{i=2}^n x_i; \\
 & \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n \leq \infty \rightarrow \langle \rangle_{i=2}^n x_i; \\
 & \quad \top
 \end{aligned}$$

## Identity

$$id(x) \equiv x$$

## Atom

$$atom(x) \equiv x \text{ is an atom} \rightarrow T;$$

$$x = \top \rightarrow \top;$$

$$x = \perp \rightarrow \perp;$$

$F$

## Equals

$$eq(x) \equiv x = \langle y, z \rangle, y, z \text{ in } \mathbf{B} \ \& \ y = z \rightarrow T;$$

$$x = \langle y, z \rangle, y, z \text{ in } \mathbf{B} \ \& \ y \neq z \rightarrow F;$$

$$[x = \langle y, z \rangle \text{ or } x = \langle y, z \rangle] \ \& \ y \not\sqsubseteq z \ \& \ z \not\sqsubseteq y \rightarrow F;$$

$$[x = \langle y, z \rangle \text{ or } x = \langle y, z \rangle] \ \& \ [y \sqsubseteq z \text{ or } z \sqsubseteq y] \rightarrow \perp;$$

$$x = \perp, \langle \rangle, \text{ or } \langle y \rangle \rightarrow \perp;$$

$\top$

## Null

$$null(x) \equiv x = \phi \rightarrow T;$$

$$x = \perp \text{ or } \langle \rangle \rightarrow \perp;$$

$$x = \top \rightarrow \top;$$

$F$

## Reverse

$$reverse(x) \equiv x = \langle \underset{i=1}{\overset{n}{x}}_i \ \& \ 0 \leq n < \infty \rightarrow \langle \underset{i=1}{\overset{n}{x}}_{n-i} \rangle;$$

$$x = \langle \underset{i=1}{\overset{n}{x}}_i \ \& \ 0 \leq n < \infty \rightarrow \langle \underset{i=1}{\overset{n}{x}} \rangle \perp;$$

$$x = \langle \underset{i=1}{\overset{\infty}{x}}_i \rangle \rightarrow \langle \underset{i=1}{\overset{\infty}{x}} \rangle \perp;$$

$$x = \perp \rightarrow \perp;$$

$\top$

## Distribute from left; distribute from right

$$distl : x \equiv x = \perp, \langle \rangle, \text{ or } \langle y \rangle \rightarrow \langle \rangle;$$

$$x = \langle y, \perp \rangle \text{ or } \langle y, \perp \rangle \rightarrow \langle \rangle;$$

$$\begin{aligned}
x &= \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y, z_i \rangle; \\
x &= \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y, z_i \rangle;
\end{aligned}$$

⊤

$$\begin{aligned}
\text{distr} : x \equiv x &= \perp, \langle \rangle, \langle \perp \rangle, \langle \langle \rangle \rangle, \text{ OR } \langle \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y_i, \perp \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y_i, \perp \rangle; \\
x &= \langle \perp, z \rangle, \langle \langle \rangle, z \rangle, \langle \perp, z \rangle, \text{ OR } \langle \langle \rangle, z \rangle \rightarrow \langle \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \ \& \ 1 \leq n < \infty \rightarrow \langle \rangle_{i=1}^n \langle y_i, z \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \ \& \ 1 \leq n \leq \infty \rightarrow \langle \rangle_{i=1}^n \langle y_i, z \rangle
\end{aligned}$$

⊤

## Length

$$\begin{aligned}
\text{length} : x \equiv x &= \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \perp; \\
x &= \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow n; \\
x &= \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp; \\
x &= \perp \rightarrow \perp;
\end{aligned}$$

⊤

## Add, subtract, multiply and divide

$$\begin{aligned}
+ : x \equiv x &= \perp \text{ OR } \langle \rangle \rightarrow \perp; \\
x &= \langle y \rangle \ \& \ y \text{ is a number OR } \perp \rightarrow \perp; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y + z; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers OR } \perp \rightarrow \perp;
\end{aligned}$$

⊤

$$\begin{aligned}
- : x \equiv x &= \perp \text{ OR } \langle \rangle \rightarrow \perp; \\
x &= \langle y \rangle \ \& \ y \text{ is a number OR } \perp \rightarrow \perp; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers} \rightarrow y - z; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers OR } \perp \rightarrow \perp;
\end{aligned}$$

⊤



$$\begin{aligned}
\times : x \equiv & x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
& x = \langle y \rangle \text{ \& } y \text{ is a number or } \perp \rightarrow \perp; \\
& x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers} \rightarrow y \times z; \\
& x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers or } \perp \rightarrow \perp; \\
& \top
\end{aligned}$$

$$\begin{aligned}
\div : x \equiv & x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
& x = \langle y \rangle \text{ \& } y \text{ is a number or } \perp \rightarrow \perp; \\
& x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers} \rightarrow y \div z; \\
& x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers or } \perp \rightarrow \perp; \\
& \top
\end{aligned}$$

## Transpose

$$\begin{aligned}
\text{trans} : x \equiv & x = \perp \rightarrow \perp; \\
& x = \langle \rangle_{i=1}^n x_i, 0 \leq n \leq \infty, \text{ and for some } m, 0 \leq m \leq \infty, \\
& x_i \in \{ \perp, \langle \rangle_{j=1}^m x_{ij} \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& x = \langle \rangle_{i=1}^n x_i, 0 \leq n < \infty, \text{ and for some } m, 0 \leq m \leq \infty, \\
& x_i \in \{ \perp, \langle \rangle_{j=1}^m x_{ij} \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& x = \langle \rangle_{i=1}^n x_i, 0 \leq n \leq \infty, \text{ and for some } m, 0 \leq m < \infty, \\
& x_i \in \{ \perp \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \text{and if } n \geq 1 \text{ there exists } k, 1 \leq k \leq n \text{ such that } x_k = \langle \rangle_{j=1}^m x_{kj} \\
& \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& x = \langle \rangle_{i=1}^n x_i, 0 \leq n < \infty, \text{ and for some } m, 0 \leq m < \infty, \\
& x_i \in \{ \perp \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \text{and if } n \geq 1 \text{ there exists } k, 1 \leq k \leq n \text{ such that } x_k = \langle \rangle_{j=1}^m x_{kj} \\
& \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& \top
\end{aligned}$$

## And, or, not

$$\begin{aligned}
 \text{and: } x \equiv & \quad x = \perp, \langle \rangle, \langle T \rangle \text{ OR } \langle F \rangle \rightarrow \perp; \\
 & \quad x = \langle T, F \rangle, \langle F, T \rangle, \text{ OR } \langle F, F \rangle \rightarrow F; \\
 & \quad x = \langle T, F \rangle, \langle F, T \rangle, \text{ OR } \langle F, F \rangle \rightarrow F; \\
 & \quad x = \langle T, T \rangle \text{ OR } \langle T, T \rangle \rightarrow T; \\
 & \quad x = \langle y, z \rangle \text{ OR } \langle y, z \rangle \& y, z \in \{\perp, T, F\} \rightarrow \perp; \\
 & \quad \top
 \end{aligned}$$

etc.

## Append left; append right

$$\begin{aligned}
 \text{apndl: } x \equiv & \quad x = \perp \text{ OR } \langle \rangle \rightarrow \langle \perp \rangle; \\
 & \quad x = \langle y \rangle \rightarrow \langle y \rangle; \\
 & \quad x = \langle y, \perp \rangle \text{ OR } \langle y, \perp \rangle \rightarrow \langle y \rangle; \\
 & \quad x = \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle y \rangle; \\
 & \quad x = \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle y \rangle; \\
 & \quad x = \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \& 1 \leq n < \infty \rightarrow \langle y, z_1 \dots z_n \rangle; \\
 & \quad x = \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \& 1 \leq n < \infty \rightarrow \langle y, z_1 \dots z_n \rangle; \\
 & \quad x = \langle y, \langle \rangle_{i=1}^\infty z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^\infty z_i \rangle \rightarrow \langle y, z_1, z_2 \dots \rangle; \\
 & \quad \top
 \end{aligned}$$

$$\begin{aligned}
 \text{apndr: } x \equiv & \quad x = \perp, \langle \rangle, \langle \perp \rangle, \langle \langle \rangle \rangle, \text{ OR } \langle \langle \rangle \rangle \rightarrow \langle \perp \rangle; \\
 & \quad x = \langle \perp, z \rangle \text{ OR } \langle \perp, z \rangle \rightarrow \langle \perp \rangle; \\
 & \quad x = \langle \langle \rangle, z \rangle \text{ OR } \langle \langle \rangle, z \rangle \rightarrow \langle \perp \rangle; \\
 & \quad x = \langle \langle \rangle, z \rangle \text{ OR } \langle \langle \rangle, z \rangle \rightarrow \langle z \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^n y_i \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^n y_i \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, z \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^\infty y_i \rangle \rightarrow \langle \langle \rangle_{i=1}^\infty y_i \rangle; \\
 & \quad x = \langle \langle \rangle_{i=1}^\infty y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^\infty y_i, z \rangle \rightarrow \langle \langle \rangle_{i=1}^\infty y_i \rangle; \\
 & \quad \top
 \end{aligned}$$

## Right selectors; right tail

$$1r : x \equiv x = \perp \rightarrow \perp;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \perp;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n < \infty \rightarrow x_n;$$

$$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp;$$

⊥

$$2r : x \equiv x = \perp \rightarrow \perp;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \perp;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 2 \leq n < \infty \rightarrow x_{n-1};$$

$$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp;$$

⊥

$$tlr : x \equiv x = \perp \rightarrow \langle \rangle;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \langle \rangle_{i=1}^{n-1} x_i;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \langle \rangle_{i=1}^{n-1} x_i;$$

$$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \langle \rangle_{i=1}^{\infty} x_i;$$

⊥

etc.

## Rotate left; rotate right

$$rotl : x \equiv x = \perp \text{ OR } \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle x_1 \rangle \rightarrow \langle \perp \rangle;$$

$$x = \langle x_1 \rangle \rightarrow \langle x_1 \rangle;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 2 \leq n < \infty \rightarrow \langle x_2 \dots x_n, \perp \rangle;$$

$$x = \langle \rangle_{i=1}^n x_i \ \& \ 2 \leq n < \infty \rightarrow \langle x_2 \dots x_n, x_1 \rangle;$$

$$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \langle \rangle_{i=2}^{\infty} x_i;$$

⊥

$$rotr : x \equiv x = \perp \text{ OR } \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle x_1 \rangle \rightarrow \langle \perp \rangle;$$

$$\begin{aligned}
x = \langle x_1 \rangle &\rightarrow \langle x_1 \rangle; \\
x = \langle \rangle_{i=1}^n x_i \ \& \ 2 \leq n < \infty \rightarrow \langle \perp, x_1 \dots x_{n-1} \rangle; \\
x = \langle \rangle_{i=1}^n x_i \ \& \ 2 \leq n < \infty \rightarrow \langle x_n, x_1 \dots x_{n-1} \rangle; \\
x = \langle \rangle_{i=1}^{\infty} x_i &\rightarrow \langle \perp, x_1, x_2, \dots \rangle; \\
\top
\end{aligned}$$

### Composition

$$(f \circ g)(x) \equiv f(g(x))$$

### Construction

$$[f_1, \dots, f_n](x) \equiv \langle f_1(x), \dots, f_n(x) \rangle$$

### Condition

$$\begin{aligned}
(p \rightarrow f; g)(x) &\equiv p(x) = T \rightarrow f(x); \\
& p(x) = F \rightarrow g(x); \\
& p(x) = \perp \rightarrow \perp; \\
\top
\end{aligned}$$

### Constant

$$\bar{x}(y) \equiv y = \top \rightarrow \top;$$

$x$

### Insert

Right Insert (from the original FP Insert)

$$\begin{aligned}
/f(x) &\equiv x = \perp \rightarrow \perp; \\
x = \langle x_1 \rangle &\rightarrow x_1; \\
x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f(\langle x_1, /f(\langle x_2, \dots, x_n \rangle) \rangle); \\
x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 0 \rightarrow \perp; \\
\top
\end{aligned}$$

Left Insert

$$\begin{aligned}
\backslash f(x) &\equiv x = \perp \rightarrow \perp; \\
x = \langle x_1 \rangle &\rightarrow x_1; \\
x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f(\langle \backslash f(\langle x_1, \dots, x_{n-1} \rangle), x_n \rangle); \\
x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 0 \rightarrow \perp; \\
\top
\end{aligned}$$

**Apply to all**

$$\alpha f(x) \equiv x = \perp \rightarrow \langle \rangle;$$

$$x = \phi \rightarrow \phi;$$

$$x = \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

⊥

**Binary to unary**

$$(bu f x)(y) \equiv f(\langle x, y \rangle)$$

**While**

$$(while p f)(x) \equiv p(x) = T \rightarrow (while p f)(f(x));$$

$$p(x) = F \rightarrow x;$$

⊥

Appendix C  
The SFP Primitive Functions and Functional Form Operators  
Operational Semantics

This appendix contains the rewrite rules for the operational semantics of SFP. For each primitive function or functional form, a list of rules is given. In addition to the stated restrictions, each rule carries an implicit restriction that none of the previous rules for that primitive function or functional form apply.

**Selector**

$$\begin{aligned}
l(x) \equiv & \quad x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n < \infty \rightarrow x_1; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n \leq \infty \rightarrow x_1; \\
& \quad x \text{ is a constant} \rightarrow \top
\end{aligned}$$

and for any positive integer  $s$

$$\begin{aligned}
s(x) \equiv & \quad x = \perp \rightarrow \perp; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ s \leq n < \infty \rightarrow x_s; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < s \rightarrow \perp; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ s \leq n \leq \infty \rightarrow x_s; \\
& \quad x \text{ is a constant} \rightarrow \top
\end{aligned}$$

**Tail**

$$\begin{aligned}
tl(x) \equiv & \quad x = \perp \rightarrow \perp; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \langle \rangle_{i=2}^n x_i; \\
& \quad x = \langle \rangle_{i=1}^n x_i \ \& \ 1 \leq n \leq \infty \rightarrow \langle \rangle_{i=2}^n x_i; \\
& \quad x \text{ is a constant} \rightarrow \top
\end{aligned}$$

## Identity

$$id(x) \equiv x$$

## Atom

$$atom(x) \equiv x \text{ is an atom} \rightarrow T;$$

$$x = \top \rightarrow \top;$$

$$x = \perp \rightarrow \perp;$$

$$x \text{ is a constant} \rightarrow F$$

## Equals

$$eq(x) \equiv x = \langle y, z \rangle, y, z \text{ in } \mathbf{B} \ \& \ y = z \rightarrow T;$$

$$x = \langle y, z \rangle, y, z \text{ in } \mathbf{B} \ \& \ y \neq z \rightarrow F;$$

$$[x = \langle y, z \rangle \text{ or } x = \langle y, z \rangle] \ \& \ y \not\sqsubseteq z \ \& \ z \not\sqsubseteq y \rightarrow F;$$

$$[x = \langle y, z \rangle \text{ or } x = \langle y, z \rangle] \ \& \ [y \sqsubseteq z \text{ or } z \sqsubseteq y] \rightarrow \perp;$$

$$x = \perp, \langle \rangle, \text{ or } \langle y \rangle \rightarrow \perp;$$

$$x \text{ is a constant} \rightarrow \top$$

## Null

$$null(x) \equiv x = \phi \rightarrow T;$$

$$x = \perp \text{ or } \langle \rangle \rightarrow \perp;$$

$$x = \top \rightarrow \top;$$

$$x \text{ is a constant} \rightarrow F$$

## Reverse

$$reverse(x) \equiv x = \langle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \langle_{i=1}^n x_{n-i};$$

$$x = \langle_{i=1}^n \rangle x_i \ \& \ 0 \leq n < \infty \rightarrow \langle_{i=1}^n \rangle \perp;$$

$$x = \langle_{i=1}^{\infty} x_i \rightarrow \langle_{i=1}^{\infty} \rangle \perp;$$

$$x = \perp \rightarrow \perp;$$

$$x \text{ is a constant} \rightarrow \top$$

## Distribute from left; distribute from right

$$distl : x \equiv x = \perp, \langle \rangle, \text{ or } \langle y \rangle \rightarrow \langle \rangle;$$

$$x = \langle y, \perp \rangle \text{ or } \langle y, \perp \rangle \rightarrow \langle \rangle;$$

$$\begin{aligned}
x &= \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle y, \langle \rangle \rangle \text{ OR } \langle y, \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y, z_i \rangle; \\
x &= \langle y, \langle \rangle_{i=1}^n z_i \rangle \text{ OR } \langle y, \langle \rangle_{i=1}^n z_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y, z_i \rangle; \\
x &\text{ is a constant } \rightarrow \top
\end{aligned}$$

$$\begin{aligned}
\text{distr} : x \equiv & x = \perp, \langle \rangle, \langle \perp \rangle, \langle \langle \rangle \rangle, \text{ OR } \langle \langle \rangle \rangle \rightarrow \langle \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y_i, \perp \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i \rangle \rightarrow \langle \rangle_{i=1}^n \langle y_i, \perp \rangle; \\
x &= \langle \perp, z \rangle, \langle \langle \rangle, z \rangle, \langle \perp, z \rangle, \text{ OR } \langle \langle \rangle, z \rangle \rightarrow \langle \rangle; \\
x &= \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \ \& \ 1 \leq n < \infty \rightarrow \langle \rangle_{i=1}^n \langle y_i, z \rangle \\
x &= \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ OR } \langle \langle \rangle_{i=1}^n y_i, z \rangle \ \& \ 1 \leq n \leq \infty \rightarrow \langle \rangle_{i=1}^n \langle y_i, z \rangle \\
x &\text{ is a constant } \rightarrow \top
\end{aligned}$$

## Length

$$\begin{aligned}
\text{length} : x \equiv & x = \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow \perp; \\
x &= \langle \rangle_{i=1}^n x_i \ \& \ 0 \leq n < \infty \rightarrow n; \\
x &= \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp; \\
x &= \perp \rightarrow \perp; \\
x &\text{ is a constant } \rightarrow \top
\end{aligned}$$

## Add, subtract, multiply and divide

$$\begin{aligned}
+ : x \equiv & x = \perp \text{ OR } \langle \rangle \rightarrow \perp; \\
x &= \langle y \rangle \ \& \ y \text{ is a number OR } \perp \rightarrow \perp; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers } \rightarrow y + z; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers OR } \perp \rightarrow \perp; \\
x &\text{ is a constant } \rightarrow \top \\
- : x \equiv & x = \perp \text{ OR } \langle \rangle \rightarrow \perp; \\
x &= \langle y \rangle \ \& \ y \text{ is a number OR } \perp \rightarrow \perp; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers } \rightarrow y - z; \\
x &= \langle y, z \rangle \text{ OR } \langle y, z \rangle \ \& \ y, z \text{ are numbers OR } \perp \rightarrow \perp; \\
x &\text{ is a constant } \rightarrow \top
\end{aligned}$$



$$\begin{aligned}
\times : x \equiv & \quad x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
& \quad x = \langle y \rangle \text{ \& } y \text{ is a number or } \perp \rightarrow \perp; \\
& \quad x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers} \rightarrow y \times z; \\
& \quad x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers or } \perp \rightarrow \perp; \\
& \quad x \text{ is a constant} \rightarrow \top \\
\div : x \equiv & \quad x = \perp \text{ or } \langle \rangle \rightarrow \perp; \\
& \quad x = \langle y \rangle \text{ \& } y \text{ is a number or } \perp \rightarrow \perp; \\
& \quad x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers} \rightarrow y \div z; \\
& \quad x = \langle y, z \rangle \text{ or } \langle y, z \rangle \text{ \& } y, z \text{ are numbers or } \perp \rightarrow \perp; \\
& \quad x \text{ is a constant} \rightarrow \top
\end{aligned}$$

## Transpose

$$\begin{aligned}
trans : x \equiv & \quad x = \perp \rightarrow \perp; \\
& \quad x = \langle \rangle_{i=1}^n x_i, \quad 0 \leq n \leq \infty, \text{ and for some } m, \quad 0 \leq m \leq \infty, \\
& \quad x_i \in \{ \perp, \langle \rangle_{j=1}^m x_{ij} \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \quad \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& \quad x = \langle \rangle_{i=1}^n x_i, \quad 0 \leq n < \infty, \text{ and for some } m, \quad 0 \leq m \leq \infty, \\
& \quad x_i \in \{ \perp, \langle \rangle_{j=1}^m x_{ij} \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \quad \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& \quad x = \langle \rangle_{i=1}^n x_i, \quad 0 \leq n \leq \infty, \text{ and for some } m, \quad 0 \leq m < \infty, \\
& \quad x_i \in \{ \perp \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \quad \text{and if } n \geq 1 \text{ there exists } k, \quad 1 \leq k \leq n \text{ such that } x_k = \langle \rangle_{j=1}^m x_{kj} \\
& \quad \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& \quad x = \langle \rangle_{i=1}^n x_i, \quad 0 \leq n < \infty, \text{ and for some } m, \quad 0 \leq m < \infty, \\
& \quad x_i \in \{ \perp \} \cup \{ \langle \rangle_{j=1}^p x_{ij} \mid 0 \leq p \leq m \} \\
& \quad \text{and if } n \geq 1 \text{ there exists } k, \quad 1 \leq k \leq n \text{ such that } x_k = \langle \rangle_{j=1}^m x_{kj} \\
& \quad \rightarrow \langle \rangle_{j=1}^m \langle \rangle_{i=1}^n y_{ij}, \text{ where } y_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ exists;} \\ \perp, & \text{otherwise.} \end{cases} \\
& \quad x \text{ is a constant} \rightarrow \top
\end{aligned}$$

## And, or, not

$$\begin{aligned}
 \text{and} : x \equiv & \quad x = \perp, \langle \downarrow \rangle, \langle T \downarrow \rangle \text{ or } \langle F \downarrow \rangle \rightarrow \perp; \\
 & \quad x = \langle T, F \downarrow \rangle, \langle F, T \downarrow \rangle, \text{ or } \langle F, F \downarrow \rangle \rightarrow F; \\
 & \quad x = \langle T, F \rangle, \langle F, T \rangle, \text{ or } \langle F, F \rangle \rightarrow F; \\
 & \quad x = \langle T, T \downarrow \rangle \text{ or } \langle T, T \rangle \rightarrow T; \\
 & \quad x = \langle y, z \downarrow \rangle \text{ or } \langle y, z \rangle \ \& \ y, z \in \{\perp, T, F\} \rightarrow \perp; \\
 & \quad x \text{ is a constant} \rightarrow \top
 \end{aligned}$$

etc.

## Append left; append right

$$\begin{aligned}
 \text{apndl} : x \equiv & \quad x = \perp \text{ or } \langle \downarrow \rangle \rightarrow \langle \perp \downarrow \rangle; \\
 & \quad x = \langle y \downarrow \rangle \rightarrow \langle y \downarrow \rangle; \\
 & \quad x = \langle y, \perp \downarrow \rangle \text{ or } \langle y, \perp \rangle \rightarrow \langle y \downarrow \rangle; \\
 & \quad x = \langle y, \langle \downarrow \downarrow \rangle \rangle \text{ or } \langle y, \langle \downarrow \rangle \rangle \rightarrow \langle y \downarrow \rangle; \\
 & \quad x = \langle y, \langle \rangle \downarrow \rangle \text{ or } \langle y, \langle \rangle \rangle \rightarrow \langle y \rangle; \\
 & \quad x = \langle y, \langle \downarrow_{i=1}^n z_i \downarrow \rangle \rangle \text{ or } \langle y, \langle \downarrow_{i=1}^n z_i \rangle \rangle \ \& \ 1 \leq n < \infty \rightarrow \langle y, z_1 \dots z_n \downarrow \rangle; \\
 & \quad x = \langle y, \langle \downarrow_{i=1}^{\infty} z_i \downarrow \rangle \rangle \text{ or } \langle y, \langle \downarrow_{i=1}^{\infty} z_i \rangle \rangle \ \& \ 1 \leq n < \infty \rightarrow \langle y, z_1 \dots z_n \rangle; \\
 & \quad x = \langle y, \langle \downarrow_{i=1}^{\infty} z_i \downarrow \rangle \rangle \text{ or } \langle y, \langle \downarrow_{i=1}^{\infty} z_i \rangle \rangle \rightarrow \langle y, z_1, z_2 \dots \rangle; \\
 & \quad x \text{ is a constant} \rightarrow \top
 \end{aligned}$$

In addition, the following rules are used with *apndl*:

$$\text{apndl} : \langle x, y \rangle \rightarrow \langle x \sim y \rangle$$

$$\text{apndl} : \langle x, y \downarrow \rangle \rightarrow \langle x \sim y \downarrow \rangle$$

$$\langle \text{list}_1 \sim \langle \text{list}_2 \rangle \rangle \rightarrow \langle \text{list}_1, \text{list}_2 \rangle$$

$$\langle \text{list}_1 \sim \langle \text{list}_2 \rangle \downarrow \rangle \rightarrow \langle \text{list}_1, \text{list}_2 \downarrow \rangle$$

$$\langle \text{list}_1 \sim \langle \text{list}_2 \downarrow \rangle \rangle \rightarrow \langle \text{list}_1, \text{list}_2 \downarrow \rangle$$

$$\langle \text{list}_1 \sim \langle \text{list}_2 \downarrow \downarrow \rangle \rangle \rightarrow \langle \text{list}_1, \text{list}_2 \downarrow \downarrow \rangle$$

$$\langle \text{list} \sim \perp \rangle \rightarrow \langle \text{list} \downarrow \rangle$$

$$\langle \text{list} \sim \perp \downarrow \rangle \rightarrow \langle \text{list} \downarrow \downarrow \rangle$$

$\langle list \sim a \rangle \rightarrow \top$ , where  $a$  is an atom

$\langle list \sim a \rangle \rightarrow \top$ , where  $a$  is an atom

$\langle list \sim \top \rangle \rightarrow \top$

$\langle list \sim \top \rangle \rightarrow \top$

$apndr : x \equiv x = \perp, \langle \rangle, \langle \perp \rangle, \langle \langle \rangle \rangle, \text{ or } \langle \langle \rangle \rangle \rightarrow \langle \perp \rangle;$

$x = \langle \perp, z \rangle \text{ or } \langle \perp, z \rangle \rightarrow \langle \perp \rangle;$

$x = \langle \langle \rangle, z \rangle \text{ or } \langle \langle \rangle, z \rangle \rightarrow \langle \perp \rangle;$

$x = \langle \langle \rangle, z \rangle \text{ or } \langle \langle \rangle, z \rangle \rightarrow \langle z \rangle;$

$x = \langle \langle \rangle_{i=1}^n y_i \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle;$

$x = \langle \langle \rangle_{i=1}^n y_i \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle;$

$x = \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ or } \langle \langle \rangle_{i=1}^n y_i, z \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, \perp \rangle;$

$x = \langle \langle \rangle_{i=1}^n y_i, z \rangle \text{ or } \langle \langle \rangle_{i=1}^n y_i, z \rangle \& 1 \leq n < \infty \rightarrow \langle y_1 \dots y_n, z \rangle;$

$x = \langle \langle \rangle_{i=1}^{\infty} y_i \rangle \rightarrow \langle \rangle_{i=1}^{\infty} y_i;$

$x = \langle \langle \rangle_{i=1}^{\infty} y_i, z \rangle \text{ or } \langle \langle \rangle_{i=1}^{\infty} y_i, z \rangle \rightarrow \langle \rangle_{i=1}^{\infty} y_i;$

$x$  is a constant  $\rightarrow \top$

## Right selectors; right tail

$1r : x \equiv x = \perp \rightarrow \perp;$

$x = \langle \rangle_{i=1}^n x_i \& 0 \leq n < \infty \rightarrow \perp;$

$x = \langle \rangle_{i=1}^n x_i \& 1 \leq n < \infty \rightarrow x_n;$

$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp;$

$x$  is a constant  $\rightarrow \top$

$2r : x \equiv x = \perp \rightarrow \perp;$

$x = \langle \rangle_{i=1}^n x_i \& 0 \leq n < \infty \rightarrow \perp;$

$x = \langle \rangle_{i=1}^n x_i \& 2 \leq n < \infty \rightarrow x_{n-1};$

$x = \langle \rangle_{i=1}^{\infty} x_i \rightarrow \perp;$

$x$  is a constant  $\rightarrow \top$

$tlr : x \equiv x = \perp \rightarrow \langle \rangle;$

$x = \langle \rangle_{i=1}^n x_i \& 0 \leq n < \infty \rightarrow \langle \rangle_{i=1}^{n-1} x_i;$

$$\begin{aligned}
x = \langle \underset{i=1}{\overset{n}{\rangle}} x_i \ \& \ 0 \leq n < \infty & \rightarrow \langle \underset{i=1}{\overset{n-1}{\rangle}} x_i; \\
x = \langle \underset{i=1}{\overset{\infty}{\rangle}} x_i & \rightarrow \langle \underset{i=1}{\overset{\infty}{\rangle}} x_i; \\
x \text{ is a constant} & \rightarrow \top
\end{aligned}$$

etc.

### Rotate left; rotate right

$$\begin{aligned}
\text{rotl} : x \equiv x = \perp \text{ or } \langle \rangle \rightarrow \langle \rangle \\
x = \langle \rangle & \rightarrow \langle \rangle; \\
x = \langle x_1 \rangle & \rightarrow \langle \perp \rangle; \\
x = \langle x_1 \rangle & \rightarrow \langle x_1 \rangle; \\
x = \langle \underset{i=1}{\overset{n}{\rangle}} x_i \ \& \ 2 \leq n < \infty & \rightarrow \langle x_2 \dots x_n, \perp \rangle; \\
x = \langle \underset{i=1}{\overset{n}{\rangle}} x_i \ \& \ 2 \leq n < \infty & \rightarrow \langle x_2 \dots x_n, x_1 \rangle; \\
x = \langle \underset{i=1}{\overset{\infty}{\rangle}} x_i & \rightarrow \langle \underset{i=2}{\overset{\infty}{\rangle}} x_i; \\
x \text{ is a constant} & \rightarrow \top
\end{aligned}$$

$$\begin{aligned}
\text{rotr} : x \equiv x = \perp \text{ or } \langle \rangle \rightarrow \langle \rangle \\
x = \langle \rangle & \rightarrow \langle \rangle; \\
x = \langle x_1 \rangle & \rightarrow \langle \perp \rangle; \\
x = \langle x_1 \rangle & \rightarrow \langle x_1 \rangle; \\
x = \langle \underset{i=1}{\overset{n}{\rangle}} x_i \ \& \ 2 \leq n < \infty & \rightarrow \langle \perp, x_1 \dots x_{n-1} \rangle; \\
x = \langle \underset{i=1}{\overset{n}{\rangle}} x_i \ \& \ 2 \leq n < \infty & \rightarrow \langle x_n, x_1 \dots x_{n-1} \rangle; \\
x = \langle \underset{i=1}{\overset{\infty}{\rangle}} x_i & \rightarrow \langle \perp, x_1, x_2, \dots \rangle; \\
x \text{ is a constant} & \rightarrow \top
\end{aligned}$$

### Miscellaneous Rules

In the following, let  $x_i = \top$  for some  $i$ ,  $1 \leq i \leq n$ , in  $\langle x_1, \dots, x_n \rangle$  and  $\langle x_1, \dots, x_n \rangle$  :

$$\langle x_1 \dots x_n \rangle \rightarrow \top$$

$$\langle x_1 \dots x_n \rangle \rightarrow \top$$

### Composition

$$(f \circ g)(x) \equiv f(g(x))$$

## Construction

$$[f_1, \dots, f_n](x) \equiv \langle f_1(x), \dots, f_n(x) \rangle$$

## Condition

$$(p \rightarrow f; g)(x) \equiv p(x) = T \rightarrow f(x);$$

$$p(x) = F \rightarrow g(x);$$

$$p(x) = \perp \rightarrow \perp;$$

⊤

## Constant

$$\bar{x}(y) \equiv y = \top \rightarrow \top;$$

$x$

## Insert

Right Insert (from the original FP Insert)

$$/f(x) \equiv x = \perp \rightarrow \perp;$$

$$x = \langle x_1 \rangle \rightarrow x_1;$$

$$x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f(\langle x_1, /f(\langle x_2, \dots, x_n \rangle) \rangle);$$

$$x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 0 \rightarrow \perp;$$

⊤

Left Insert

$$\backslash f(x) \equiv x = \perp \rightarrow \perp;$$

$$x = \langle x_1 \rangle \rightarrow x_1;$$

$$x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 2 \rightarrow f(\langle \backslash f(\langle x_1, \dots, x_{n-1} \rangle), x_n \rangle);$$

$$x = \langle x_1, \dots, x_n \rangle \ \& \ n \geq 0 \rightarrow \perp;$$

⊤

Apply to all

$$\alpha f(x) \equiv x = \perp \rightarrow \langle \rangle;$$

$$x = \phi \rightarrow \phi;$$

$$x = \langle \rangle \rightarrow \langle \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

$$x = \langle x_1, \dots, x_n \rangle \rightarrow \langle f(x_1), \dots, f(x_n) \rangle;$$

⊤

## Binary to unary

$$(bu\ f\ x)(y) \equiv f(\langle x, y \rangle)$$

## While

$$(while\ p\ f)(x) \equiv p(x) = T \rightarrow (while\ p\ f)(f(x));$$

$$p(x) = F \rightarrow x;$$

⊥

## BIBLIOGRAPHY

- [Arvind *et al.* 1982] Arvind, J. D. Brock, and K. Pingali. FP 1.5: Backus' FP with Higher Order Functions. Computation Structures Group Note 46, MIT Laboratory for Computer Science, March, 1982.
- [Backus 1978] Backus, J. Can programming be liberated from the vonNeumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21, No. 8, Aug 1978, pp. 613-641.
- [Backus *et al.* 1986] Backus, J., J. Williams, & E. Wimmers. FL Language Manual. Technical Report RJ5339(54809). IBM, 1986.
- [Backus 1981] Backus, J. Is Computer Science Based on the Wrong Fundamental Concept of 'Program'? An Extended Concept, in *Algorithmic Languages*. North-Holland Publishing Company, 1981. pp. 133-165. Ed. by de Bakker & van Vliet.
- [Broy 1982] Broy, M. A fixed point approach to applicative multiprogramming, in *Theoretical Foundations of Programming Methodology*. D. Reidel, 1982. pp. 565-622. Ed. by M. Broy & G. Schmidt.
- [Burge 1975] Burge, W.H. *Recursive programming techniques*. Reading, Mass: Addison-Wesley, 1975.
- [Dosch & Moller 1984] Dosch, W. & B. Moller. Busy and lazy FP with infinite objects. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, 1984. pp. 282-292.
- [Feldman 1982] Feldman, G. Functional Specifications of a Text Editor. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, 1982. pp. 37-46.
- [Halpern *et al.* 1985] Halpern, J., J. Williams, E. Wimmers & T. Winkler. Denotational Semantics and Rewrite Rules for FP. In *Conference Record of the*

- Twelfth Annual ACM Symposium on Principles of Programming Languages*, 1985. pp. 108–120.
- [Halpern *et al.* 1986] Halpern, J., J. Williams, & E. Wimmers. Completeness of Rewrite Rules and Rewrite Strategies for FP. Technical Report RJ5099(52999). IBM, 1986.
- [Henderson 1982] Henderson, P. *Purely Functional Operating Systems in Functional Programming and its Applications*. Cambridge University Press, 1982. pp. 177–192.
- [Ida & Tanaka 1983] Ida, T. & J. Tanaka. Functional programming with streams, in *Information Processing*. North-Holland, 1983. pp. 265–270.
- [Jacobson 1974] Jacobson, N. *Basic Algebra I*. San Francisco: W.H. Freeman & Co., 1974.
- [Kahn 1974] Kahn, G. The semantics of a simple language for parallel programming. In Proceedings, IFIP Congress 74, 1974 pp. 471–475.
- [Kahn & MacQueen 1977] Kahn, G. & D.B. MacQueen. Coroutines and networks of parallel processes. In Proceedings, IFIP Congress 77, North-Holland, Jan 1977.
- [Keller 1977] Keller, R.M. Semantics of parallel program graphs. Technical Report UUCS-77-110, University of Utah, July 22, 1977.
- [Keller 1978] Keller, R.M. Denotational models for parallel programs with indeterminate operators, in *Formal Description of Programming Concepts*. North-Holland, 1978. pp. 337–366.
- [Kieburtz & Shultis 1981] Kieburtz R.B. & J. Shultis. Transformations of FP Program Schemes. In *ACM-MIT Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture.*, Oct 1981. pp. 41–48.



- [Kleene 1952] Kleene, S.C. *Introduction to Meta-mathematics*. Princeton, N.J.: D. Van Nostrand, Inc., 1952.
- [Magó 1979] Magó, G.A. A network of microprocessors to execute reduction languages. (Two parts). *International Journal of Computer and Information Sciences* 8, 5 (1979) pp. 349–385, 8, 6 (1979) pp. 435–471.
- [Magó 1980] Magó, G.A. A cellular computer architecture for functional programming. Digest of Papers, *IEEE Computer Society COMPCON* (Spring 1980), pp. 179–187.
- [Magó & Middleton 1984] Magó, G.A. and D. Middleton: "The FFP Machine - -A Progress Report". *International Workshop on High-Level Computer Architecture 84*, Los Angeles, California, May 23-25, 1984.
- [Milner 1984] Milner, R. A proposal for Standard ML. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, 1984. pp. 184–197.
- [Myers 1980] Myers, T.J. Infinite Structures in Programming Languages. Ph.D. Thesis, University of Pennsylvania, 1980.
- [Nakata & Sassa 1983] Nakata, I. & M. Sassa. Programming with streams. Technical Report RJ3751(43317). IBM, Jan 1983.
- [Nickl 1985] Nickl, F. A Denotational Semantics for Backus' FP with Infinite Objects, Technical Report of the Fakultät für Mathematik und Informatik, Universität Passau, 1985.
- [Scott 1971] Scott, D.S. the Lattice of Flow Diagrams. In *Proceedings of the Symposium on the Semantics of Algorithmic Languages*, E. Engeler, editor. Springer-Verlag, Berlin, 1971. pp. 311–366.
- [Scott 1972] Scott, D.S. Lattice Theory, Data Types and Semantics. In *NYU Symposium on Formal Semantics*, R. Rustin, editor. Prentice-Hall, New York, 1972. pp. 64–106. G. Schmidt.

- [Scott 1986] Scott, D.S. Data types as lattices. In the *SIAM Journal on Computing*, 5 (1976). pp. 522–587.
- [Scott 1982] Scott, D.S. Lectures on a mathematical theory of computation, in *Theoretical Foundations of Programming Methodology*. D. Reidel, 1982. pp. 146–292. Ed. by M. Broy & G. Schmidt.
- [Scott & Strachey 1981] Scott, D.S. & C. Strachey. Toward a Mathematical Semantics for Computer Languages. In *Proceedings of the Symposium on Computers and Automata*, J. Fox, editor. Polytechnic Institute of Brooklyn Press, New York, 1971. pp. 19–46.
- [Stoy 1977] Stoy, J.E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, Mass: The MIT Press, 1977.
- [Stoy 1982] Stoy, J.E. Some Mathematical Aspects of Functional Programming, in *Functional Programming and its Applications*. Cambridge University Press, 1982. pp. 217–252.
- [Strachey 1966] Strachey, C. Towards a Formal Semantics, in *Formal Language Description Languages for Computer Programming*, T.B. Steel, editor. North-Holland, Amsterdam, 1966. pp. 198–220.
- [Tanaka & Ida 1981] Tanaka, J. & T. Ida. Stream extension for fp-like language. 1981. (Unpublished paper.)
- [Tarski 1955] Tarski, A. A lattice-theoretical fixpoint theorem and its Applications. In *Pacific Journal of Mathematics*, 5, 1955.
- [Thomas & Stanat 1985] Thomas, T.A. & D.F. Stanat. An FP Domain with Infinite Objects. In *Proceedings of the Mathematical Foundations of Programming Semantics*, Lecture Notes in Computer Science No. 239, Springer-Verlag, 1985.
- [Turner 1982] Turner, D.A. Recursion equations as a programming language, in *Functional Programming and its Applications*. Cambridge University Press, 1982. pp. 1–28. Ed. by J. Darlington, P. Henderson, & D.A. Turner.

- [Wadler 1981] Wadler, P. Applicative style of programming, program transformation and list operators. In *ACM-MIT Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture.*, Oct 1981. pp. 25-32.
- [Wadler 1984] Wadler, P. Listlessness is Better Than Laziness: Lazy Evaluation and Garbage Collection at Compile-Time. In *Proceedings ACM Symposium on LISP and Functional Programming*, August 1984.
- [Williams & Wimmers 1988] Williams, J. & E. Wimmers. Sacrificing simplicity for convenience: Where do you draw the line? In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988. pp. 169-179.