

October 1989

Report No. STAN-CS-89-1288

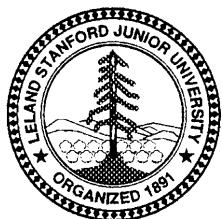
**Programming and Proving
with
Function and Control Abstractions**

by

Carolyn Talcott

Department of Computer Science

**Stanford University
Stanford, California 94305**



REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION		1 b RESTRICTIVE MARKINGS			
2a SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION /AVAILABILITY OF REPORT			
2b DECLASSIFICATION / DOWNGRADING SCHEDULE					
4 PERFORMING ORGANIZATION REPORT NUMBER(S) <i>Stan - CS - 89 - 1288</i>		5 MONITORING ORGANIZATION REPORT NUMBER(S)			
6a NAME OF PERFORMING ORGANIZATION <i>Computer Science Dept</i>	6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION			
6c ADDRESS (City, State, and ZIP Code) <i>Stanford CA 94305</i>	7b ADDRESS (City, State, and ZIP Code)				
8a NAME OF FUNDING /SPONSORING ORGANIZATION <i>DARPA / IS TO</i>	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER <i>W00039-84-C-0211</i>			
10c ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

Programming and proving with function and control abstractions

12 PERSONAL AUTHOR(S)

Carynne Talcott

13a TYPE OF REPORT	13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT (Year, Month, Day) 1 9 Oct 1989	15 PAGE COUNT <i>118</i>
--------------------	---	--	-----------------------------

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) <i>program equivalence, stream, coroutine, function abstraction, object behavior, control abstraction, derived program</i>
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

see other side

20 DISTRIBUTION /AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION
22a NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)
	22c OFFICE SYMBOL

Rum is an intensional semantic theory of function and control abstractions as computation primitives. It is a mathematical foundation for understanding and improving current practice in symbolic (Lisp-style) computation. The theory provides, in a single context, a variety of semantics ranging from structures and rules for carrying out computations to an interpretation as functions on the computation domain. Properties of powerful programming tools such as functions as values, streams, aspects of object oriented programming, escape mechanisms, and coroutines can be represented naturally. In addition a wide variety of operations on programs can be treated including program transformations which introduce function and control abstractions, compiling morphisms that transform control abstractions into function abstractions, and operations that transform intensional properties of programs into extensional properties. The theory goes beyond a theory of functions computed by programs, providing tools for treating both intensional and extensional properties of programs. This provides operations on programs with meanings to transform as well as meanings to preserve. Applications of this theory include expressing and proving properties of particular programs and of classes of programs and studying mathematical properties of computation mechanisms. Additional applications are the design and implementation of interactive computation systems and the mechanization of reasoning about computation.

These notes are based on lectures given at the Western Institute of Computer Science summer program, 31 July – 1 August 1986. Here we focus on programming and proving with function and control abstractions and present a variety of example programs, properties, and techniques for proving these properties.

Programming and proving
with
function and control abstractions

Carolyn Talcott

**Computer Science Department, Stanford University
Stanford, CA, 94305**
CLT@SAIL.STANFORD.EDU

Copyright © 1989 by Carolyn Talcott

Research supported by ARPA contract N00039-84-C-0211.

Table Of Contents

1.	Introduction	1
1.1.	About <i>Rum</i>	1
1.2.	An annotated reading list	3
1.2.1.	Programming with function and control abstractions.	3
1.2.2.	Semantics	4
2.	Programming with function and control abstractions.	6
2.1.	Multiple values and functionals	7
2.2.	Object behaviors.	8
2.3.	Computing number tree products.	9
2.3.1.	Simple recursion on number trees.	10
2.3.2.	Improving by pruning unnecessary computation.	10
2.3.3.	Continuation passing computation of the tree product.	11
2.3.4.	Pruning by context noting and switching.	12
2.4.	Streams as infinite sequences.	15
2.5.	Using coroutines to transform sequences.	16
3.	The meta world.	21
3.1.	Finite Sequences.	21
3.2.	Finite Maps.	22
3.3.	Inductive generation.	22
3.4.	The S-expression data structure.	23
4.	The Rum world.	25
4.1.	Domains of <i>Rum</i>	25
4.2.	Operations and relations.	28
4.2.1.	Computation rules.	29
4.2.2.	Computation sequences.	30
4.3.	Abbreviations and notation.	32
4.4.	Operational relations	33
5.	Towards a theory of function and control abstractions	37
5.1.	The language and its semantics	37

Table of Contents

5.2.	The basic theory	39
5.2.1.	Logic of partial terms and approximation	39
5.2.2.	In laws	41
5.2.3.	Some proof schemes	42
5.3.	Computational laws	43
5.3.1.	Extensionality and recursion	45
5.4.	Some simple derived laws	46
5.5.	Puzzling with current puzzle.	49
5.6.	Context motion	51
5.6.1.	Context motion theorem	51
5.6.2.	Consequences of the context motion theorem	53
6.	Proving properties	56
6.1.	Vari-ary functions and function schemes	56
6.2.	Object behaviors.	59
6.3.	The tree product pfns	60
6.4.	Streams and coroutines.	65
6.4.1.	Streams	65
6.4.2.	Coroutines	68
6.4.3.	Remarks	72
7.	Derived properties and programs	73
7.1.	Examples of simple derived properties	73
7.2.	Definition of simple derived property	74
7.3.	Derived programs	76
7.4.	Analysis of tree product computations.	80
7.4.1.	Analysis of the recursive tree product pfm	81
7.4.2.	Analysis of the escaping tree product pfn	86
7.5.	Possible elaborations	88
8.	Conclusions	89
9.	References	92
10.	Appendix: Proofs and technical miscellaney	98
10.1.	Properties of operational relations	98
10.1.1.	A refinement of operational approximation and equivalence	98
10.1.2.	Proofs of easy consequences	99
10.1.3.	Substitution	100

Table of Contents

10.1.4. Extensionality and recursion	102
10.2. Proofs of some simple derived laws	104
10.3. Proof of context motion theorem	107
10.4. Representation of Computation	113
10.5. Relation to standard definition of operational relations	114

List of Figures

Figure 1.	The S-expression data structure	23
Figure 2.	<i>Rum Domains</i>	25
Figure 3.	Domain equations	26
Figure 4.	Relations and operations of <i>Rum</i>	28
Figure 5.	Rules for stepping	30
Figure 6.	Example computation sequence	31
Figure 7.	Evaluated position contexts	52
Figure 8.	Example computation sequence	74
Figure 9.	Derivation map action on forms	79

1. Introduction

Rum is an intensional semantic theory of function and control abstractions as computation primitives. It is a mathematical foundation for understanding and improving current practice in symbolic (Lisp-style) computation. The theory provides, in a single context, a variety of semantics ranging from structures and rules for carrying out computations to an interpretation **as** functions on the computation domain. Properties of powerful programming tools such **as** functions as values, streams, aspects of object oriented programming, escape mechanisms, and coroutines can be represented naturally. In addition a wide variety of operations on programs can be treated including program transformations which introduce function and control abstractions, compiling morphisms that transform control abstractions into function abstractions, and operations that transform intensional properties of programs into extensional properties. The theory goes beyond a theory of functions computed by programs, providing tools for treating both intensional and extensional properties of programs. This provides operations on programs with meanings to transform as well as meanings to preserve. Applications of this theory include expressing and proving properties of particular programs and of classes of programs and studying mathematical properties of computation mechanisms. Additional applications are the design and implementation of interactive computation systems and the mechanization of reasoning about computation.

These notes are based on lectures given at the Western Institute of Computer Science summer program, 31 July – 1 August 1986. Here we focus on programming and proving with function and control abstractions and present a variety of example programs, properties, and techniques for proving these properties.

1.1. About *Rum*

In the Lisp community, people traditionally speak of vanilla Lisp when referring to the pure first-order fragment which has a simple interpretation in terms of partial functions on S-expressions. Following this tradition (and being fond of sum-raisin ice cream) we have chosen to call our flavor of Lisp rum, and we use *Rum* to refer to the theory we have developed about this flavor. Rum flavored Lisp is very much like the Scheme dialect of Lisp [Steele and Sussman 1975], [Rees and Clinger 1986].

The theory *Rum* is based on an intensional semantic theory of function and control abstractions as computation primitives. An intensional semantic theory is a semantics based on a view of computation as a process of generating computation structures and an interpretation of programs as descriptions of computations. Computation structures and rules for generating them determine an abstract machine for carrying out computations, thus providing an operational semantics. The

operational approximation and equivalence relations induced by the operational semantics provide a mathematical interpretation of the informal concept of “programs as black boxes”. Roughly two programs are operationally equivalent if they can not be distinguished by any computational context. The operational relations are preserved by substitution and abstraction and the recursion operator (*Y*-combinator analog) computes the least **fixed** point of a functional with respect to operational approximation. Thus we have also an extensional semantics.

Intensional properties of programs can be represented as properties of computation structures. Some examples are the number of multiplications executed, the number of context switches, the maximum stack depth required in a computation, and the trace of a computation. Extensional properties of programs are properties of the “function” computed. They can be expressed in terms of the operational approximation and equivalence relations. Some examples of extensional properties are notions of equality for streams and cosoutines and characterizations of functionals implementing strategies for searching tree-structured spaces.

An abstraction formalizes the notion of an object being given uniformly in terms of some parameter. Application of an abstraction instantiates the parameter to the argument of the application. Function and control abstractions are mathematically tractable computation primitives that allow one to express easily a wide variety of computation mechanisms and programming styles and to reason naturally about programs that employ these primitives. Function abstractions describe the computation of (partial) functions. A function abstraction produced by evaluation of a lambda expression contains information giving the interpretation of free variables occurring in the expression in the creation time environment. In addition to ordinary functions, function abstractions can be used to represent structured (possibly infinite) data such as tuples and streams, to represent the continuation of a computation, to describe delayed or lazy evaluation, and (when assignment abstractions are added) to represent objects in the object-oriented style of computation. Control abstractions represent contexts built up in the process of carrying out a computation. They provide a means of suspending and resuming computations and can be used to program non-local control mechanisms such as escaping and co-soutining.

These notes are organized as follows. The second part of this introduction is an annotated list of suggestions for further reading. §2 presents examples that illustrate the use of multiple values and function and control abstractions to describe a variety of computation mechanisms. It includes sample computations and informal discussion of properties of the example programs and their computations. §3 summarizes the basic mathematical tools and notation that will be used. In §4 the intensional semantic theory is presented, the operational relations that are the semantic basis for expressing properties of programs are defined, and key properties are stated. In §5 a language for expressing properties of *Rum* programs

is defined, its semantics is explained, some basic axioms and rules for proving assertions are stated, and additional are laws derived as examples and tools for later use. §6 contains precise statements and proofs of the extensional properties of programs discussed in §2. In §7 methods for treating intensional properties are developed and illustrated. In §8 we summarize what we have done and discuss some further developments and extensions of the theory. The appendix (§10) contains proofs of theorems stated in §4 and other miscellaneous technical details.

1.2. An annotated reading list

This list is divided into two parts – roughly practice and theory – and each list is sorted alphabetically. The list is included here, as it also serves as a mini survey of concepts and results that form the background for *Rum* and of related work. Additional discussion of some of these papers can be found in [Talcott 1985; Chapter II].

1.2.1. Programming with function and control abstractions.

- [Abelson and Sussman 1985] Text for introductory programming course at MIT, based on the language Scheme. Operational viewpoint - programs are viewed as descriptions of computation.
- [Friedman, D. P. et.al. 1984] A basic text on programming in Scheme. Programs are viewed as functions. There is much emphasis on the use of various forms of abstraction (function, control, textual, . . .) in programming.
- [Burge 1971] Many examples of programming with functions as arguments and values.
- [Burge 1975rec] A basic text on writing recursive programs and using functions as arguments and values.
- [Burge 1975str] Examples of programming with streams illustrating the use of streams in a variety of applications. Contains a veritable library of operations on streams.
- [Burstall 1968] Using functionals and the Landin *J* (for jump) operator to program search strategies. (See entry for Landin below.)
- [Conway 1963] Origin of the notion of coroutine as a way of organizing a large compilation into small independent processes.
- [Henderson 1980] Treats many aspects of functional programming including use of higher order functions, abstract machines, implementation issues, and many excellent programming examples.

- [Kahn and McQueen 1977] A language for defining dynamically configurable networks of processes. An important subset of the language can be interpreted sequentially (as coroutines) or as synchronized parallel processes without affecting the behavior. The paper includes example programs and proofs of correctness.

1.2.2. Semantics

- . [Basendregt 1981] More than you want to know about the lambda **calculus**, but an excellent place to look for definitions, concepts, statements and proofs of theorems, etc.
- [Felleisen and Friedman 1986,7; Felleisen 1987] Extensions of Plotkin's call-by-value lambda calculus (see Plotkin entry) with operations for capturing and aborting the current computation context (control calculus [FELLEISEN AND FRIEDMAN 1986]) and with labeled values, assignment and derefesencing operations (assignment calculus [FELLEISEN AND FRIEDMAN 1987]). The calculi are derived from operational semantics given by abstract machines in the spirit of Landin (see Landin entry). These calculi have Church-Rosses theorems, Standardization theorems and can be used as the basis for reasoning about programs that use function and control abstractions and assignment.
- [Landin 1964–6] An operational semantics for AE (the language of the lambda calculus) and for IAE (AE plus non-local control and assignment constructs) given in terms of the SECD machine (an abstract machine for evaluating expressions.) In Landin[1965], this semantics is used to provide a semantics for Algol 60 expressions by translation to IAE expressions. In Landin[1966], Iswim (If you See What I Mean – IAE plus “syntactic sugar” defining additional important computation primitives) is proposed as the basis of an approach to language design. Iswim is a framework providing mechanisms for naming things (binding) and for defining functional relations, thus seducing language design to a matten of choosing primitive data and operations and choosing printed and physical representations.
- [Plotkin 1975] Defines a call-by-value lambda calculus with constants and shows that the equational semantics given by lambda-v seduction and lambda-v equivalence is compatible with the operational semantics (evaluation relation) given by Landin's SECD machine. The notion of operational (black-box) equivalence is defined in terms of the evaluation relation and shown to properly contain lambda-v equivalence.
- [Schmidt 1986] A general text on denotational semantics.
- [Scott 1976] Descsibes, in great detail, the graph model of the lambda calculus (discovered independently by Scott and Plotkin). The graph model is an example of a general method for constructing mathematical models of the

lambda calculus. Expressions in the language of the lambda calculus are interpreted as “graphs” of continuous functions on a domain with a complete partial order. Equations provable in classical lambda calculus and many more equations hold in this model. The Y combinator computes the least-fixed-point operator on graphs of functions. Domain theory is developed within the model using lambda-definable retractions to represent domains and the fixed point operator to solve domain equations.

- . [Scott and Strachey 1971] Original presentation of basic methods of Scott-Strachey semantics including clearly worked out examples.

2. Programming with function and control abstractions.

In this section we give examples of programs that use function and control abstractions and multiple values to describe a variety of computation tools including program schemes, object behaviors, escape and coroutine mechanisms, and infinite streams. We begin with an informal summary of notation and concepts used in the examples. This together with the discussion of the examples is (hopefully) sufficient to make the main points clear. The precise syntax and semantics is given in §4 and many of the properties stated informally here will be stated more precisely and proved in later sections.

The computation domain contains data and data operations from the underlying data structure, *pfns*, and *continuations*. Functions computed are “variary” – their domains and ranges are finite sequences of elements from the computation domain. • I is the empty sequence and $[v_0, v_1]$ is the concatenation of the sequences v_0 and v_1 . Elements of the computation domain are treated as singleton sequences.

The underlying data structure for our examples is an S-expression structure. The domain is closed under pairing and contains integers, strings, and a special constant **Nil** representing the empty list. The data operations include (using Lisp names) pairing and projections (**Cons**, **Car**, **Cdr**); tests for atoms (non-pairs) and zero (Atom, Zerop); successor, addition, and multiplication (Addl, +, *); and the string operations (**StrMk**, **StrUn**). For S-expressions a and b we write $a \cdot b$ for the value of **Cons**(a , b). Lists are generated as usual from the empty list by pairing. We write $\langle a_1, \dots, a_n \rangle$ for $a_1 \cdot (\dots \cdot (a_n \cdot \text{Nil}))$. **StrMk** takes a sequence of integers (character codes) and returns the corresponding string. **StrUn** takes a string and returns the corresponding sequence of character codes. **#a** is the character code for the letter ‘a’ and we write “abc” for the value of **StrMk**[#a, #b, #c].

Pfns are functional abstractions that can be thought of as partial functions containing information describing how the value of the function is to be computed. Pfns correspond to Landin’s closures [Landin 1964] – the value of a lambda abstraction is a pfn containing the lambda abstraction and the values of its free variables in the evaluation environment. Continuations are control abstractions that represent computation contexts built up in the process of carrying out computations. A computation context contains the information prescribing how the computation is to continue when the current subcomputation is complete. As objects of the computation domain, pfns provide a means of passing procedures as parameters, of encapsulating an expression for later evaluation, and of constructing specialized versions a procedure by instantiating some of its parameters. Continuations provide a means of remembering and switching contexts, and of suspending and resuming computations.

We often use systems of recursion equations to define pfns. These equations are written using the sign \leftarrow and specify that the equation should be used to replace applications of the defined pfn by the body suitably instantiated.

Extensional properties of programs (properties of programs viewed as black boxes) are expressed using the *operational equivalence* relation (\cong). Two expressions or values are operationally equivalent if they can not be distinguished by any computational context. This means that an expression or **value** can be replaced by an equivalent one in any computational context without changing the meaning of the whole expression. Equivalent expressions either both escape, are both undefined or are both defined with equivalent values. Operational equivalence is preserved by substitution and abstraction and it satisfies the usual laws for conditional expressions (**if**) and the equational laws of the underlying data structure. The **stepwise** computation rules can be formulated as laws of operational equivalence. Pfn_s defined by recursion equations also satisfy the equations obtained by replacing \leftarrow by \cong . We will see many other interesting properties of operational equivalence as we proceed. If U is a set of sequences from the computation domain and u is a member of U we say that u has *sort* U . We use the term *sort* rather than type since the sorts may be organized into many different type structures and we don't wish to specify a particular one.

One final notational remark. With sequences and lambda abstraction we have two forms of expressing multi-ary functions and function application: using sequences as in $(\lambda[a, b].\varphi)[a, b] \cong \varphi$; and curried lambda application as in $(\lambda u.\lambda v.\varphi)(u, v) \cong \varphi$. In the former a, b range over are singletons and in the latter u, v range over arbitrary sequences. Note that unary application of a function to a sequence $f[x_1, \dots, x_n]$ and curried application $f(x_1, \dots, x_n)$ are different. For example $(\lambda[x, y].\varphi)[[a, b], c]$ binds x to a and y to $[b, c]$ while $(X(x, y).\varphi)([a, b], c)$ binds x to $[a, b]$ and y to b .

2.1. Multiple values and functionals.

To illustrate the expressive power of multiple arguments and values we define the string concatenation operation **StrConc** directly in terms of the data operations **StrUn** and **StrMk**. The trick is to use sequence concatenation to do the main work.

▷ **StrConc**[x, y] \leftarrow **StrMk**[**StrUn**(x), **StrUn**(y)]

For example

StrConc["abc", "def "] \cong "abcdef".

■(**Properties of StrConc**) **StrConc** maps pairs of strings to strings, is associative and has the empty string as left and right identity.

Now we look at an example where higher order functions can be thought of as program schemes (describing classes of programs). Properties of such functions can serve as proof schemes for proving properties of particular instances. $\text{Sit}(f, b, x)$ iterates a binary function f along a sequence x using b as the initial second argument.

D $\text{Sit}(f, b, x) \leftarrow \text{if}(x, f(\text{fst}(x), \text{Sit}(f, b, \text{rst}(x))), b)$

where **if** tests for the empty sequence and **fst**, **rst** compute the first and rest of a sequence.

■ **(The sort of Sit)** Let A, B be subsets of the computation domain, let f compute a total function from $A \times B$ to B and let b be an element of B . Then $\lambda x.\text{Sit}(f, b, x)$ computes a total function from A^* (the set of sequences from A) to B .

A sequence of S-expressions is converted to a list by mapping the pairing function along that sequence beginning with the empty list. Thus we can define **ListMk** by

D $\text{ListMk } k(x) \leftarrow \text{Sit}(\lambda x. \text{Cons}[x, \text{Nil}], x)$

ListMk formalizes the conventional list notation. For example we have

$\text{ListMk}[0, 1, 2] = \langle 0 1 2 \rangle$

$\text{ListMk}[a_0, \dots, a_k] = \langle a_0 \dots a_k \rangle$

■ **(The sort of ListMk)** From the sort of **Sit** it is easy to see that **ListMk** computes a total function from sequences of S-expressions to lists.

2.2. Object behaviors.

Objects in the Small-talk sense are active entities with internal state that can receive and reply to some collection of messages. The internal state of an object can only be changed by the object. This is a form of abstraction (as in abstract data type) that insures that the representation of internal state is not visible. The observable behavior of an object is its set of possible message-reply sequences. The abstraction criteria means that the representation of an objects internal state can be freely changed without changing the observable behavior of the object.

Although we can not model the sharing aspects for objects whose internal state may change, we can represent an objects behavior as a pfn which when applied to a message returns a new pfn describing the behavior of the object after receiving that message, together with its reply to that message. The behavior represented

by such a pfn is the set of possible message-reply sequences (ignoring the behavior pfn part of each reply, which is the functional representation of internal state).

As an example we consider the simplest non-trivial behavior — that of a (memory) cell. A cell's internal state is its contents. A cell with contents a accepts get and set messages. In the case of a get message, the cell replies with a and remains a cell with contents a . In the case of a set message, with contents component b the cell becomes a cell with contents b (the reply if any is just acknowledgement). For simplicity we assume any other messages are no-ops — the internal state is not changed and no reply is made. We assume there are tests **Getmsg** and **Setmsg** to distinguish message types, and in the case of set messages a selector **Contents** that extracts the contents component of the message. The pfn *Cell(u)* is defined by

```
D Cell(u) ← λm.if(Getmsg(m), [Cell(u), a],
                     if(Setmsg(m), Cell(Contents(m)),
                        Cell(a)))
```

■ **(Specifying cell behavior)** The pfn *Cell(u)* describes the behavior of a cell with contents a .

2.3. Computing number tree products.

The theme for this set of examples is the problem of computing the product of numbers in a number tree. A number tree x is either a number, or a pair of number trees $x_0 \cdot x_1$. The tree product is the product of the numbers in a number tree. The number of cells in a number tree is the number of **conses** used to construct it and the number of nodes is the number of cells plus the number of leaves. For example, 2 and $(3 \cdot 2) \cdot 2$ are number trees, the tree product of $(3 \cdot 2) \cdot 2$ is 12, the number of cells in $(3 \cdot 2) \cdot 2$ is 2, and the number of nodes in $(3 \cdot 2) \cdot 2$ is 5.

We will consider three pfns that compute the tree product function. **Tp** computes the product by the obvious recursion on the number tree structure. **Tpc** and **Tps** compute the product as one would in a hand computation — by abandoning the normal processing of a number tree when a zero is encountered and immediately returning the value zero to the caller. **Tpc** uses function abstraction and **Tps** uses control abstraction to implement the “immediately return zero to the caller” strategy.

2.3.1. Simple recursion on number trees.

Tp is defined by

$$\text{D} \quad Tp(x) \leftarrow \text{if}(\text{Atom}(x), x, Tp(\text{Car}(x)) * Tp(\text{Cdr}(x))).$$

This equation together with laws for operational equivalence and facts about data operations can be used to compute the value of $Tp(x)$ for any number tree x in the usual way. As an example we compute the value of $Tp((3 \cdot 2) \cdot 0)$.

$$\begin{aligned} Tp((3 \cdot 2) \cdot 0) &\cong \text{if}(\text{Atom}((3 \cdot 2) \cdot 0), (3 \cdot 2) \cdot 0, Tp(\text{Car}((3 \cdot 2) \cdot 0)) * Tp(\text{Cdr}((3 \cdot 2) \cdot 0))) \\ &\quad \text{;; using the definition of } Tp \\ &\cong Tp(\text{Car}((3 \cdot 2) \cdot 0)) * Tp(\text{Cdr}((3 \cdot 2) \cdot 0)) \quad \text{;; } \text{Atom}((3 \cdot 2) \cdot 0) \text{ is false} \\ &\cong Tp(3 \cdot 2) * Tp(\text{Cdr}((3 \cdot 2) \cdot 0)) \quad \text{;; } \text{Car}((3 \cdot 2) \cdot 0) \cong 3 \cdot 2 \\ &\cong (3 * 2) * Tp(\text{Cdr}((3 \cdot 2) \cdot 0)) \quad \text{;; recursively computing } Tp(3 \cdot 2) \cong 3 * 2 \\ &\cong 6 * Tp(\text{Cdr}((3 \cdot 2) \cdot 0)) \quad \text{;; } 3 * 2 \cong 6 \\ &\cong 6 * Tp(0) \quad \text{;; } \text{Cdr}((3 \cdot 2) \cdot 0) \cong 0 \\ &\cong 6 * 0 \quad \text{;; computing } Tp(0) \cong 0 \\ &\cong 0 \quad \text{;; } 6 * 0 \cong 0 \end{aligned}$$

■ **(Properties of the computation of Tp)** In the above computation Car and Cdr are each applied twice and $*$ is applied twice. More generally, we can show that for any number tree x in the computation of the value of $Tp(x)$ the number of applications of Car and Cdr is the number of non-root nodes in x and the number $\boxed{\Delta X}$ applications of $*$ is the number of cells in x .

2.3.2. Improving by pruning unnecessary computation.

If zero occurs in a number tree then the product will be zero. Thus we can make the following requirement for a pfn computing the tree product:

(‡) *if a zero is encountered in traversing the number tree, terminate traversal and return zero without further computation.*

One way to define a pfn meeting this specification is to define an auxiliary pfn which has an additional parameter used to store context information. The key is to insure that as each node of the number tree is visited the initial calling context and the current number tree context (what remains to be done in processing the number tree) are both available. Then computation can continue normally or abort processing and return a value directly to the calling context.

2.3.3. Continuation passing computation of the tree product.

Tpc uses an auxiliary pfn **Tc** that describes a continuation passing style computation ([Reynolds 1972], [Steele and Sussman 1975]). **Tc** expects two arguments – a number tree and a continuation pfn. The continuation pfn represents the current number tree context. A key feature of continuation passing style computation is that no computation context is built up. The current computation context is the calling context, and simply returning a value from a subcomputation returns it to the calling context. To continue the computation normally, the value is returned to the continuation pfn (i.e. the continuation pfn is applied to the returned value). **Tpc** and **Tc** are defined by

$$\begin{aligned} \text{D} \quad & \mathbf{Tpc}(x) \leftarrow \mathbf{Tc}(x, \mathbf{I}) \\ \text{D} \quad & \mathbf{Tc}(x, f) \leftarrow \mathbf{if}(\mathbf{Atom}(x), \\ & \quad \mathbf{if}(\mathbf{Zerop}(x), 0, f(x)), \\ & \quad \mathbf{Tc}(\mathbf{Car}(x), \mathbf{Ta}(x, f))) \\ \text{D} \quad & \mathbf{Ta}(x, f) \leftarrow \lambda y. \mathbf{Tc}(\mathbf{Cdr}(x), \mathbf{Td}(y, f)) \\ \text{D} \quad & \mathbf{Td}(y, f) \leftarrow \lambda z. f(y * z) \end{aligned}$$

Tpc computes the tree product by calling **Tc** with the identity pfn, $\mathbf{I} = \lambda z. z$, as the initial continuation pfn. For number trees x and continuation functions f , we can analyze the computation of $\mathbf{Tc}(x, f)$ as follows. In the case x is zero, zero is returned and the computation is complete. In the case x is a non-zero atom, computation proceeds by applying f to x to carry out the remainder of the computation. In the case x is a non-atom, the left subtree **Car**(x) is processed with continuation **Ta**(x, f). **Ta**(x, f) is the pfn constructed by closing the lambda expression $\lambda y. \mathbf{Tc}(\mathbf{Cdr}(x), \mathbf{Td}(y, f))$ in the environment determined by the values of x and f . If a zero is encountered in **Car**(x) then zero will be returned as the value. If no zeros are encountered in **Car**(x), the value m_a of $\mathbf{Tp}(\mathbf{Car}(x))$ will be returned to **Ta**(x, f). Computation of $\mathbf{Ta}(x, f)(m_a)$ proceeds by processing the right subtree **Cdr**(x) with continuation pfn $\mathbf{Td}(m_a, f)$. $\mathbf{Td}(m_a, f)$ is the closure of $\lambda z. f(y * z)$ in an environment that assigns the value m_a to the symbol y and the value of f to f . If a zero is encountered in **Cdr**(x) then zero will be returned as the value. If no zeros are encountered in **Cdr**(x), the value m_d of $\mathbf{Tp}(\mathbf{Cdr}(x))$ will be returned to $\mathbf{Td}(m_a, f)$. Computation of $\mathbf{Td}(m_a, f)(m_d)$ proceeds by multiplying m_a and m_d and returning the result to f . As an example we compute the value of $\mathbf{Tpc}((3 . 2) * 0)$.

$$\begin{aligned} \mathbf{Tpc}((3 . 2) * 0) &\cong \mathbf{Tc}((3 . 2) * 0, \mathbf{I}) \quad ;; \text{definition } \mathbf{Tpc} \\ &\cong \mathbf{Tc}(3 . 2, \mathbf{Ta}((3 . 2) * 0, \mathbf{I})) \end{aligned}$$

```

;; definition Tc, Atom((3 . 2) . 0) is false, Car((3 . 2) . 0)  $\cong$  3 . 2
 $\cong \text{Tc}(3, \text{Ta}(3 . 2, \text{Ta}((3 . 2) . 0, \text{I})))$ 

;; definition Tc, Atom(3 . 2) is false, Car(3 . 2)  $\cong$  3
 $\cong \text{Ta}(3 . 2, \text{Ta}((3 . 2) . 0, \text{I}))(3)$ 

;; definition Tc, Atom(3) is true, Zerop(3) is false
 $\cong \text{Tc}(2, \text{Td}(3, \text{Ta}((3 . 2) . 0, \text{I}))) \quad \text{;; definition Ta, Cdr}(3 . 2) \cong 2$ 
 $\cong \text{Td}(3, \text{Ta}((3 . 2) . 0, \text{I}))(2)$ 

;; definition Tc, Atom(2) is true, Zerop(2) is false
 $\cong \text{Ta}((3 . 2) . 0, \text{I}))(6) \quad \text{;; definition Td, } 3 * 2 \cong 6$ 
 $\cong \text{Tc}(0, \text{Td}(6, \text{I})) \quad \text{;; definition Ta, Cdr}((3 . 2) . 0) \cong 0$ 
 $\cong 0 \quad \text{;; definition Tc, Atom}(0) \text{ is true, Zerop}(0) \text{ is true}$ 

```

■ **(The functions computed by **Tc** and **Tpc**)** The function computed by **Tc** is characterized by the following equation

$$(\text{Tc}.\text{Tp}) \quad \text{Tc}(x, f) \cong \text{if}(\text{Inz}(x), 0, f(\text{Tp}(x)))$$

where x ranges over number trees and $\text{Inz}(x)$ is true iff zero occurs in x . $(\text{Tc}.\text{Tp})$ can be proved by number tree induction using the informal analysis of **Tc** given above. From $(\text{Tc}.\text{Tp})$ and properties of the identity pfn it is easy to see that **Tpc** computes the tree product function.

$$(\text{Tpc}.\text{Tp}) \quad \text{Tpc}(x) \cong \text{Tp}(x)$$

■ **(Properties of the computation of **Tpc**)** In the above computation of the value of $\text{Tpc}((3 . 2) . 0)$, **Car** and **Cdr** are each applied twice and $*$ is applied once. More generally, we can show that for any number tree x , in the computation of the value of $\text{Tpc}(x)$ the number of applications of **Car** and **Cdr** is number of nodes before the leftmost zero in the depthfirst traversal of x and the number of applications of $*$ is the number of cells in x to the left of the leftmost zero.

2.3.4. Pruning by context noting and switching.

The definition of **Tps** uses context noting to remember the calling context and context switching to return a value to the calling context. To understand context noting and switching in general, we can think of a context as an expression with a hole in it. To evaluate an expression we find the current subexpression and process that, treating the remainder of the expression as the current context. For example

in $\text{Tp}(\text{Car}(x)) * \text{Tp}(\text{Cdr}(x))$ the current subexpression is $\text{Car}(x)$ and the current context is $\text{Tp}(\{\dots\}) * \text{Tp}(\text{Cdr}(x))$. Computation proceeds by replacing $\text{Car}(x)$ by its value say x_a . The current subexpression then becomes $\text{Tp}(x_a)$ and the current context becomes $\{\dots\} * \text{Tp}(\text{Cdr}(x))$. If the current subexpression is $\text{note}(c)\varphi$ then φ is evaluated in the current context with c replaced by the continuation representing that context. If the current subexpression is the application of a continuation to a value we place the value in the hole of the context represented by the continuation and evaluate the resulting expression (thus switching contexts and discarding the previous current context).

TpS notes its calling context and applies the auxiliary pfn Ts to its number tree argument and the noted context. Ts computes the product in essentially the same manner that Tp does. Its additional argument is a continuation representing the calling context. The current computation context is the current number tree context. If a zero is encountered the context parameter is applied to zero to return zero directly to the calling context. TpS and Ts are defined by

$$\begin{aligned} D \quad \text{TpS}(x) &\leftarrow \text{note}(c)\text{Ts}(x, c) \\ D \quad \text{Ts}(x, c) &\leftarrow \text{if}(\text{Atom}(x), \\ &\quad \text{if}(\text{Zerop}(x), c(0), x), \\ &\quad \text{Ts}(\text{Car}(x), c) * \text{Ts}(\text{Cdr}(x), c)) \end{aligned}$$

Three properties of context noting and switching are needed compute the value of $\text{TpS}(x)$:

- (note.abs) If two expressions φ_0, φ_1 are equivalent when the variable c ranges over continuations then the their note abstractions with respect to c are equivalent.
- (note.id) If the note variable c does not occur free in an expression φ then $\text{note}(c)c(\varphi) \cong \varphi$;
- (escape) If the current subexpression is the application of a continuation then we can discard the current context – replace the expression being evaluated by the current subexpression.

As an example we compute the value of $\text{TpS}((3 \cdot 2) \cdot 0)$. (note.abs) is used implicitly – working inside a note expression we may assume the **note** variable ranges over continuations and we may replace the note body by an equivalent expression.

$$\begin{aligned} \text{TpS}((3 \cdot 2) \cdot 0) &\cong \text{note}(c)\text{Ts}((3 \cdot 2) \cdot 0, c) \quad ; \text{ definition } \text{TpS} \\ &\cong \text{note}(c)(\text{Ts}(3 \cdot 2, c) * \text{Ts}(\text{Cdr}(3 \cdot 2) \cdot 0, c)) \\ &\quad ; \text{ definition } \text{Ts}, \text{Atom}((3 \cdot 2) \cdot 0) \text{ is false, } \text{Car}((3 \cdot 2) \cdot 0) \cong 3 \cdot 2 \end{aligned}$$

$$\begin{aligned}
&\cong \text{note}(c)(6 * \text{Ts}(\text{Cdr}((3 * 2) + 0), c)) \\
&\quad \text{;; computing } \text{Ts}(3 * 2, c) \cong 6 \text{ as for } \text{Tp} \\
&\cong \text{note}(c)(6 * c(0)) \quad \text{;; computing } \text{Ts}(\text{Cdr}((3 * 2) + 0), c) \cong c(0) \\
&\cong \text{note}(c)(c(0)) \quad \text{;; by (escape)} \\
&\cong 0 \quad \text{;; by (note.id)}
\end{aligned}$$

■ **(The functions computed by Ts and Tp)** The function computed by Ts is characterized by the following equation

$$(\text{Ts}.\text{Tp}) \quad \text{Ts}(x, c) \cong \text{if}(\text{Inz}(x), c(0), \text{Tp}(x))$$

where x ranges over number trees and c ranges over continuations. From this equation and properties of noting and switching it follows that Tp computes the tree product function.

$$(\text{Tp}.\text{Tp}) \quad \text{Tp}(x) \cong \text{Tp}(x)$$

As an introduction to proving properties of noting and switching we give a proof of $(\text{Tp}.\text{Tp})$. The proof uses a simple equation $(\text{Tp}.\text{Inz})$ expressing the fact that the tree product of a number tree x containing a zero is zero.

$$(\text{Tp}.\text{Inz}) \quad \text{Tp}(x) \cong \text{if}(\text{Inz}(x), 0, \text{Tp}(x))$$

We also need a further property of noting (**note.if**) that allows us to move a **note** inside an **if** when the test does not involve the note variable.

- **(note.if)** If the note variable c does not occur free in the expression φ_0 then $\text{note}(c)\text{if}(\varphi_0, \varphi_1, \varphi_2) \cong \text{if}(\varphi_0, \text{note}(c)\varphi_1, \text{note}(c)\varphi_2)$.

Proof $(\text{Tp}.\text{Tp})$:

$$\begin{aligned}
\text{Tp}(x) &\cong \text{note}(c)\text{Ts}(x, c) \quad \text{;; dfn } \text{Tp} \\
&\cong \text{note}(c)\text{if}(\text{Inz}(x), c(0), \text{Tp}(x)) \quad \text{;; } (\text{Ts}.\text{Tp}) \\
&\cong \text{if}(\text{Inz}(x), \text{note}(c)c(0), \text{note}(c)\text{Tp}(x)) \quad \text{;; (note.if)} \\
&\cong \text{if}(\text{Inz}(x), 0, \text{Tp}(x)) \quad \text{;; (note.triv)} \\
&\cong \text{Tp}(x) \quad \text{;; } (\text{Tp}.\text{Inz})
\end{aligned}$$

□ $\text{Tp}.\text{Tp}$

Notes

- Even though **Tps** uses context noting and switching in its computations the expression $\text{Tps}(x)$ has a value independent of context.
- Computation of **Tps** is “isomorphic” to computation of **Tpc**. That is, there is an essentially one-one correspondence between steps in the computation of $\text{Tpc}(x)$ and $\text{Tps}(x)$. In particular, the number of applications of any data operation is the same in both cases. The difference is that in the **Tps** case the interpreter does the work of keeping track of the computation context while in the **Tpc case** the programmer did this work.
- The definitions of **Tc** and **Ts** were obtained by applying program transformations to $(\text{Tc}.\text{Tp})$ and $(\text{Ts}.\text{Tp})$ viewed as defining equations. The transformations use standard transformation rules (see [Scherlis 1981]) augmented by rules for introducing and eliminating abstraction and for manipulating continuation application expressions. The transformation steps also constitute a proof that initial and final equations define the same function.

Endnotes

2.4. Streams as infinite sequences.

The idea of streams was introduced in [Landin 1965] to represent finite sequences whose elements are computed as they are needed rather than being computed in advance. This was needed to interpret certain iteration constructs of Algol 60. More generally, a stream is an object which when queried returns its first element together with an object computing the rest of the stream together. Thus we can think of streams as (possibly infinite) sequences whose elements can only be accessed by removing them from the sequence one at a time. In *Rum* streams are represented by pfns and are characterized by the sequences they generate. (We consider only infinite streams here, thus eliminating the need to deal with the end-of-stream case.) Note that streams are a special case of object behaviors where the only message is the ‘next element’ message.

As an example, we define the stream **Sieve** that generates the sequence of prime numbers in increasing order. The definition is based on the algorithm known as sieve of Eratosthenes.

```

D   Sieve( mt ) ← Sift( Ints(2))
D   Ints(n)(mt) ← [Ints(n + 1), n]
D   Filter(p, in)( mt ) ← let { [n, in] ← in( mt)}
                           if( Divp(p, n), Filter(p, in)( mt ), [Filter(p, in), n])
D   Sift(in)(mt) ← let{ [in, p] ← in(mt)}[Sift(Filter(p, in)), p]
```

where for numbers p, n the expression $\text{Divp}(p, n)$ is true iff p divides n . To see how Sieve works we compute the first few elements.

- (1) $\text{Sieve}(\text{mt}) \cong \text{Sift}(\text{Ints}(2))(\text{mt}) \cong [\text{Sift}(\text{Filter}(2, \text{Ints}(3))), 2]$
;; using the definition of Sieve and $\text{Ints}(2)(\text{mt}) \cong [\text{Ints}(3), 2]$
- (2) $\text{Sift}(\text{Filter}(2, \text{Ints}(3)))(\text{mt}) \cong [\text{Sift}(\text{Filter}(3, \text{Filter}(2, \text{Ints}(4)))), 3]$
;; since $\text{Filter}(2, \text{Ints}(3))(\text{mt}) \cong [\text{Filter}(2, \text{Ints}(4)), 3]$
- (3) $\text{Sift}(\text{Filter}(3, \text{Filter}(2, \text{Ints}(4))))(\text{mt}) \cong [\text{Sift}(\text{Filter}(5, \text{Filter}(3, \text{Filter}(2, \text{Ints}(6)))), 5)]$
;; using $\text{Filter}(2, \text{Ints}(4))(\text{mt}) \cong [\text{Filter}(2, \text{Ints}(6)), 5]$ and
;; $\text{Filter}(3, \text{Filter}(2, \text{Ints}(4)))(\text{mt}) \cong [\text{Filter}(3, \text{Filter}(2, \text{Ints}(5))), 5]$

■**(Properties of the sieving pfns)** Let n, p be numbers and in be a stream generating a sequence of numbers. Then

- . (i) $\text{Ints}(n)$ generates the the sequence of numbers greater than n .
- (ii) $\text{Filter}(p, \text{in})$ is a stream generating the sequence obtained by removing all multiples of p from the sequence generated by in .
- . (iii) $\text{Sift}(\text{in})$ is a stream whose first element p is the first element of in and whose remainder is the stream obtained by filtering p from in and sifting the result.

From this and a little number theory we see that **Sieve** is indeed a stream generating the sequence of prime numbers in increasing order as claimed.

Note. The **Sieve** example is derived from a description in terms of networks of processes [Kahn, G. and D. B. MacQueen 1977] which can be interpreted as coroutines or as parallel processes. It can also be represented in *Rum* as a coroutine that is, in a strong sense, equivalent to the stream version. Another good source of stream examples is [Abelson and Sussman 1985 (3.4)]. Here streams are thought of as lazy lists and represented as pairs whose first element is the next element of the stream and whose second element is function generating the remainder of the stream. **Endnote.**

2.5. Using coroutines to transform sequences.

A system of coroutines is a set of programs that interact by *resuming* one another rather than by the usual function call/return mechanism. A given coroutine when resumed will continue computation where it last left off, carry out the next portion of its computation, and then resume some other coroutine. Typically information is passed between coroutines by updating shared data structures and each coroutine has internal state that keeps track of where to begin when it is

resumed. The idea was originally presented in [Conway 1963] as a means of separating a large compiling program into a number of small independent procedures. Each procedure is a coroutine responsible for some phase in the transformation of a source program into the compiled code. Another common use of coroutines is for dealing with streams of characters in networks. As a particular example, we adapt a segment of network code from the Stanford WAITS operating system that uses coroutining (J. Weening - private communication). This code deals with a situation where one gets data in a stream of 36-bit words, but would like to see it as 8-bit bytes. There is a coroutine **INBYTE** responsible for getting the next byte from the 36-bit word stream. Another coroutine which uses the 8-bit bytes, for example to generate a 32-bit word stream, resumes **INBYTE** each time another byte is needed. **INBYTE** has nine segments of code, one for each of the nine bytes contained in two consecutive 36-bit words. To illustrate the main features we outline the machine code (based on the **DEC10** language FAIL) for a simplified version C32 that transforms 3-bit streams into 2-bit streams. Here IN is the coroutine generating the S-bit stream and OUT is the coroutine requiring a 2-bit stream. We assume 4 registers reserved as follows:

- A for control communication between C32 and IN
- B for control communication between C32 and OUT
- C for data communication between C32 and IN
- D for data communication between C32 and OUT

The instruction JSP A, (A) jumps to the location contained in A and replaces the contents of A by the location plus 1. JRST loc jumps to loc. Initially A contains the address of IN and B contains the address of C32. OUT resumes C32 by JSP B, (B). The code for C32 is

```

C32:    JSP A, (A)
C32a:   move bits CO-C1 into D0-D1
              JSP B,(B)
C32b:   move bit C2 into DO
              JSP A,(A)
C32c:   move bit CO into DI
              JSP B,(B)
C32d:   move bits C1-C2 into D0-D1
              JSP B,(B)
              JRST C32

```

In Rum we represent coroutine interaction without using internal state and updating by passing shared data and resumption points as parameters. Resumption is described by the pfn **Resume[out, x]** which resumes the context represented by out passing it a continuation representing the context in which the call occurs and the

additional parameters x . [x may be the empty sequence and hence may be omitted in calls to **Resume**.] To describe coroutine C32 we define a pfn C32 that takes a coroutine in generating a sequence of bits segmented as strings of length three and returns a coroutine generating the same sequence of bits segmented as strings of length two. Thus $C32(in)$ takes a continuation out representing the resumption point of its resumer and will eventually resume out with the next output string. Dually, when $C32(in)$ wants the next input string it will resume in and eventually be resumed with the next resumption point of in and the next input string.

```
D   C32(in)[out] ← let{ [in, w] ← Resume[in] }
    let{ [x0, x1, x2] ← StrUn(w)}
    let{ out ← Resume[out, StrMk[x0, x1]]}
    let{ [in, w] ← Resume[in] }
    let{ [x3, x4, x5] ← StrUn(w)}
    let{ out ← Resume[out, StrMk[x2, x3]]}
    let{ out ← Resume[out, StrMk[x4, x5]]}
    C32(in, out)
```

To evaluate an expression of the form $\text{let } \{[x_0, \dots, x_n] \leftarrow \varphi_{\text{arg}}\} \varphi_{\text{body}}$ first φ_{arg} is evaluated, then then x_i is associated with the i-th element of the value of φ_{arg} and φ_{body} is evaluated. Thus

$$\text{let } \{[x_0, x_1, x_2] \leftarrow \text{StrUn}("abc")\} \text{StrMk}[x_0, x_1] \cong "ab"$$

If we think of let as assignment then the above definition looks like code written in a standard imperative language.

In order to conveniently discuss properties of C32 we add “labels” naming the resumption points of C32. More precisely we define auxiliary pfns C32a, C32b, C32c, and C32d corresponding to the remainder of the computation at each resumption point. These pfns have the following properties

$$\begin{aligned} C32(in)[out] &\cong C32a(out)(\text{Resume}(in)) \\ C32a(out)[in, w] &\cong \text{let} \{[x_0, x_1, x_2] \leftarrow \text{StrUn}(w)\} \\ &\quad C32b(in, x_2)(\text{Resume}[out, \text{StrMk}[x_0, x_1]]) \\ C32b(in, x_2)[out] &\cong C32c(out, x_2)(\text{Resume}(in)) \\ C32c(out, x_2)[in, w] &\cong \text{let} \{[x_3, x_4, x_5] \leftarrow \text{StrUn}(w)\} \\ &\quad C32d(in, x_4, x_5)(\text{Resume}[out, \text{StrMk}[x_2, x_3]]) \\ C32d(in, x_4, x_5)[out] &\cong C32(in)(\text{Resume}[out, \text{StrMk}[x_4, x_5]]) \end{aligned}$$

In each case the first list of parameters (...) lists the state to be retained and the second list [...] names the components of the argument passed to the coroutine upon resumption.

The key to computing with coroutines is to note that resumptions come in pairs – the first executed by the resumer and the second by the resumee. When a coroutine g generating a sequence σ is resumed it will eventually resume its resumer with the next element of σ in a context representing the next resumption point of g . Thus $\text{Resume}(g)$ will be equivalent to $\text{Resume}(\lambda k.g'(\text{Resume}[k, z]))$ where z is next element of σ and g' is the coroutine generating the rest of the sequence. For this use of resumption the key property of Resume is given by resumption theorem (res.res). ¹

$$(res.\text{res}) \quad \text{Resume}(\lambda k.g'(\text{Resume}[k, x])) \cong [\text{top o } g', z]$$

and thus for g as above we have

$$(co) \quad \text{Resume}(g) \cong [\text{top o } g', z].$$

These properties of coroutines and resumption are all we will use.

To see how this works in our string transformation example, let in be a coroutine generating strings of length three whose first two elements are "abc" and "def" and whose remainder coroutines are in_1 and in_2 . Then

$$(In1) \quad \text{Resume}(in) \cong [\text{top o } in_1, "abc"]$$

$$(In2) \quad \text{Resume}(in_1) \cong [\text{top o } in_2, "def"]$$

We can compute the results of three successive resumptions of $C32(in)$ as follows.

$$(Res.1) \quad \text{Resume}(C32(in)) \cong \text{Resume}(\lambda k.C32(in, k))$$

$$\cong \text{Resume}(\lambda k.C32a(k)(\text{Resume}(in))) \quad ;; \text{ dfn C32}$$

$$\cong \text{Resume}(\lambda k.C32a(k)[\text{top o } in_1, "abc"]) \quad ;; (In1)$$

$$\cong \text{Resume}(\lambda k.C32b(\text{top o } in_1, \#c})(\text{Resume}[k, "ab"])) \quad ;; \text{ dfn C32a}$$

$$\cong [\text{top o } C32b(\text{top o } in_1, \#c), "ab"]$$

$$;; (\text{res.res})$$

$$(Res.2a) \quad C32b(\text{top o } in_1, \#c)(k) \cong C32c(k, \#c)(\text{Resume}(\text{top o } in_1)) \quad ;; \text{ dfn C32b}$$

¹ **top o p** converts a pfn p into a pfn that discards its context and applies p . This corresponds to the application of p at the top level.

$$\begin{aligned}
 &\cong C32c(k, \#c)[top \circ in_2, "def"] \quad;; (In2) \\
 &\cong C32d(top \circ \vartheta_2, \#e, \#f)(Resume[k, "cd"]) \quad;; \text{dfn } C32c \\
 (\text{Res.2}) \quad \text{Resume}(C32b(top \circ in_1, \#c)) &\cong [top \circ C32d(top \circ in_2, \#e, \#f), "cd"] \\
 &\quad;; (\text{Res.2a}) \text{ and } (\text{res.res})
 \end{aligned}$$

$$\begin{aligned}
 (\text{Res.3a}) \quad C32d(top \circ in_2, \#e, \#f)(k) &\cong C32(top \circ in_2)(Resume[k, "ef"]) \\
 &\quad;; \text{dfn } C32d
 \end{aligned}$$

$$\begin{aligned}
 (\text{Res.3}) \quad \text{Resume}(C32d(top \circ in_2, \#e, \#f)) &\cong [top \circ C32(top \circ in_2), "ef"] \\
 &\quad;; (\text{Res.3a}) \text{ and } (\text{res.res})
 \end{aligned}$$

Note. Streams and coroutines are alternatives for providing sequential access to data. Both provide the capability to delay computation of the next element of the sequence until it is needed. Both allow the representation of infinite sequences. They differ in the mechanism for accessing the elements of sequences (apply vs resume) and in the mechanism for remembering the internal state (noting vs lambda abstraction). Using the *Rum* representations of streams and coroutines one can define pfns that convert streams to coroutines and coroutines to streams, preserving the sequences generated. **Endnote.**

3. The meta world.

Now we review the mathematical structures and tools to be used in the definition and application of Rum. They are finite sequences, finite maps and inductive generation of domains, operations, and relations. The point is to establish notation as we assume that the reader is familiar with these notions. First some basic notation. \mathbb{N} is the set of natural numbers, 0, 1, ... and i, j, k, l, m, n will range over \mathbb{N} . For expressions e_0 and e_1 , $e_0(e_1)$ is the application of the function expressed by e_0 to e_1 . For sets A and B , $A \times B$ is the **cartesian** product of A and B with elements (a, b) for $a \in A$ and $b \in B$, and we write $A \oplus B$ for the union of A and B when A and B are disjoint. Similarly for **n-ary** products (whose elements are n-tuples) and sums. $[A \rightarrow B]$ is the set of functions from A to B and $[A \xrightarrow{P} B]$ is the set of partial functions from A to B . In the presence of partial functions, $e_0 = e_1$ ($e_0 \neq e_1$) means that both expressions e_0 and e_1 are defined and have the same value (distinct values).

As to general format, definitions are marked by the sign \square which may be followed by a name for later reference. We use $\square \text{name}$ to mark the end of a proof named **(name)**.

3.1. Finite Sequences.

For any set A , A^* is the domain of sequences from A . \square is the empty sequence and has length 0. If v_0 and v_1 are sequences then $[v_0, v_1]$ is the concatenation of v_0 and v_1 . Its length is the sum of the lengths of v_0 and v_1 . The length of a sequence v is denoted by $|v|$. Formally there is an injection map from A to the sequences of length one in A^* . Informally we will not distinguish elements of A from singleton sequences. If a is an element of A and v is a sequence then $[a, v]$ is a non-empty sequence; $1^{\text{st}}[a, v]$ is a , the first element; and $r^{\text{st}}[a, v]$ is v , the remainder. $1^{\text{st}} \square$ is \square and $r^{\text{st}} \square$ is \square . For $i < |v|$, $v \downarrow_i$ is the i^{th} element of v . In particular, $v \downarrow_0 = 1^{\text{st}} v$ and $v \downarrow_{i+1} = (r^{\text{st}} v) \downarrow_i$. Concatenation is associative with the empty sequence as right and left identity. We write $[v_1, \dots, v_n]$ for the concatenation of the sequences v_1, \dots, v_n . a is a member of v (written $a \in v$) iff v is $[v_0, a, v_1]$ for some sequences v_0, v_1 .

We distinguish $A \times A$ from the subset of A^* consisting of sequences length 2. This is because we wish to talk conveniently about multi-ary operations whose argument domains may be sequence domains. Thus (v_1, \dots, v_n) (as in $f(v_1, \dots, v_n)$) is an n-tuple of sequences and is not the same as $[v_1, \dots, v_n]$. From the tuple each v_i can be retrieved intact while in the concatenation expression they become part of a single sequence.

3.2. Finite Maps.

Finite maps are functions with finite domains. We represent such maps as finite sets of argument value pairs called bindings. For sets A and B , $[A \xrightarrow{f} B]$ is the set of finite maps from A to B . It is generated from the empty map $\{\}$ by the binding operation. The domain of the empty map is the empty set \emptyset (also written $\{\}$). For $a \in A$, $b \in B$, and ξ a finite map from A to B , $\xi\{a \leftarrow b\}$ is **the** map obtained from ξ by binding b to a . The domain of $\xi\{a \leftarrow b\}$ is obtained from the domain of ξ by adding a . For a' in the domain of $\xi\{a \leftarrow b\}$ we **have**

$$\xi\{a \leftarrow b\}(a') = \begin{cases} b & \text{if } a' = a \\ \xi(a') & \text{if } a' \neq a. \end{cases}$$

Two finite maps are equal just when they are equal as functions. Different constructions may give rise to the same functions since the order in which bindings of different elements of the domain are added does not matter and old bindings are forgotten.

3.3. Inductive generation.

Finite inductive generation is our main tool for defining domains, operations and relations. An inductively generated domain is defined by giving rules for constructing elements of the domains and rules for determining equality. The domains contain only the elements obtained by finitely many constructions. Elements generated by different constructions are different unless they can be proved equal using the rules for equality. Abstractly these domains are essentially initial algebras where the signature of the algebra can be read from the construction rules. For such domains we have principles for definition and proof by induction on the generation of objects in the domain. For example the domain of finite maps from A to B is an inductively generated domain. There are two constructors, the empty map and the binding operation, equality of maps is equality of the corresponding functions, and the application laws for finite maps and the definition of the domain of a finite map are defined by finite map induction.

Operations and relations may also be defined by inductive generation. For this purpose we think of operations as just a special kind of relation. Relations are defined by giving a set of rules for determining whether a given tuple is in the relation. Formally the defined relation is the least relation (set of tuples) satisfying the closure conditions expressed by the rules. The rules are often presented as logical implications. For inductively defined relations there are also corresponding principles of proof by induction.

We refer the reader to Feferman [1982] for a theory of finite inductive definitions, to Goguen and Meseguer [1983] for more about initial algebras, and to Moschovakis [1975] or Aczel [1977] for a general theory of inductive definitions.

3.4. The S-expression data structure.

Computation in *Rum* is defined relative to a fixed but arbitrary data structure \mathfrak{D} . We assume only that \mathfrak{D} is a structure $(\mathbf{D}, \mathbf{O}^{\mathbf{D}})$ where \mathbf{D} is the domain of data elements and $\mathbf{O}^{\mathbf{D}}$ is a set of data operations mapping \mathbf{D}^* to \mathbf{D}^* . The S-expression data structure $\mathfrak{D}_{\text{sexp}} = (\mathbf{D}_{\text{sexp}}, \mathbf{O}^{\mathbf{D}_{\text{sexp}}})$ is typical of the data structures we have in mind. It contains a variety of data construction primitives and provides an abstraction of the algebraic aspects of data structures commonly used in symbolic computation. The S-expression domains and operations are summarized in Figure 1.

Sort	Constructor	Constructor Domain	Recognizer	Unconstructor
\mathbf{D}_{zero}	ZeroMk	\mathbf{M}_t	Zerop	
\mathbf{D}_{neg}	Sub1	$\mathbf{D}_{\text{neg}} \oplus \mathbf{D}_{\text{zero}}$	Negp	Add1
\mathbf{D}_{pos}	Add1	$\mathbf{D}_{\text{pos}} \oplus \mathbf{D}_{\text{zero}}$	Posp	Sub1
\mathbf{D}_{str}	StrMk	$\mathbf{D}_{\text{int}}^*$	Strp	StrUn
\mathbf{D}_{mtl}	MtIMk	\mathbf{M}_t	Mtp	
\mathbf{D}_{pair}	PairMk	$\mathbf{D}_{\text{sexp}} \times \mathbf{D}_{\text{sexp}}$	Pairp	PairUn

Figure 1. The S-expression data structure

The elements of the S-expression domain \mathbf{D}_{sexp} are of four sorts: \mathbf{D}_{mtl} , \mathbf{D}_{int} , \mathbf{D}_{str} , and \mathbf{D}_{pair} . \mathbf{D}_{mtl} contains a single object, the empty list; the elements of \mathbf{D}_{int} are the integers; the elements of \mathbf{D}_{str} are strings of integers; and \mathbf{D}_{pair} consists of pairs of S-expressions. To describe the generation of \mathbf{D}_{sexp} , we split \mathbf{D}_{int} into three sorts: \mathbf{D}_{neg} – the negative integers; \mathbf{D}_{zero} – the integer 0; and \mathbf{D}_{pos} – the positive integers.

The primitive S-expression operations $\mathbf{O}^{\mathbf{D}_{\text{sexp}}}$ are constructors, unconstructors, and recognizers for each of the sorts. \mathbf{D}_{sexp} is freely generated by the construction operations applied to their construction domains. \mathbf{D}_{zero} is generated by ZeroMk applied to the empty sequence; \mathbf{D}_{neg} is generated by Sub1 applied to non-positive integers; and \mathbf{D}_{pos} is generated by Add1 applied to non-negative integers. \mathbf{D}_{mtl} is generated by MtIMk applied to the empty sequence; \mathbf{D}_{str} is generated by StrMk applied to sequences of integers; and \mathbf{D}_{pair} is generated by PairMk applied to S-expression sequences of length 2.

An unconstructor applied to an element of the corresponding sort returns the sequence from which that element was constructed. **PairUn** is the unconstructor for pairs and **StrUn** is the unconstructor for strings. **Add1** serves as the unconstructor for negative integers, and **Sub1** as the unconstructor for positive integers. [Unconstructors for singleton domains are omitted.]

A recognizer applied to an element of the sort that it recognizes returns that element. The recognizer for **D_{neg}** is **Negp**; for **D_{zero}** is **Zerop**; for **D_{pos}** is **Posp**; for **D_{str}** is **Strp**; for **D_{mtl}** is **Mtlp**; and for **D_{pair}** is **Pairp**.

A constructor applied to a sequence not in its construction domain and an unconstructor or recognizer applied to anything other than a data element of the corresponding sort return the empty sequence.

These basic operations are sufficient to define any computable function on the S-expression domain. For simplicity we will assume some additional operations are given. These include standard arithmetic operations such as + and *, standard projection operations on pairs **Car** and **Cdr**, and **Atom** the negation of **Pairp**. To conform to traditional Lisp notation we will use **Cons** as another name for **PairMk** and will also use **Nil** to denote the empty list.

We will use the usual notation for integers and pairs strings and lists. Thus **ZeroMk(0)** = 0, **Add1(0)** = 1, **Sub1(0)** = -1, and **PairMk[a, b]** = $a \cdot b$. #a, #b, ..., denote integer codes for the characters 'a', 'b', etc. and **StrMk[#a, #b, #c]** = "abc". Lists are the subset of S-expressions generated from the empty list <> by pairing arbitrary S-expressions with lists. Thus **MtlMk(0)** = <> and **<a₁ . . . a_n>** abbreviates $a_1 \cdot (\dots (a_n \cdot <>) \dots)$ as usual.

The following equations illustrate the laws for data operations:

$\text{Pairp}(a_0 \cdot a_1) = a_0 \cdot a_1$	$\text{PairUn}(a_0 \cdot a_1) = [a_0, a_1]$
$\text{Pairp}("abc") = \square$	$\text{PairUn}("abc") = \square$
$\text{Mtlp}(<a>) = \square$	$\text{Mtlp}(<>) = <>$
$\text{Add1}(\text{Sub1}(0)) = 0$	$\text{Add1}(\square) = \bullet 1$
$\text{StrUn}("abc") = [\#a, \#b, \#c]$	

4. The *Rum* world.

Rum is a theory of computation developed with the goal of treating both intensional and extensional aspects of programs that use functional and control abstractions. Many of the ideas have analogs in [Landin 1964,5,6], [Reynolds 1972], [Wegner 1971,2], and [Steele and Sussman 1975]. In this section we define the basic domains, operations, and relations and introduce some further notation. More detailed developments of the computation theory and proofs of theorems can be found in Talcott[1985].

We assume that a set of symbols $\$y$, a data domain \mathbf{D} , and set of data operations $\mathbf{O}^{\mathbf{D}}$ acting on sequences from \mathbf{D} are given. The S-expression structure described in §3 is an example of a data domain and set of data operations.

4.1. Domains of *Rum*.

Figure 2 lists the domains (sorts of objects) of *Rum* together with metavariables ranging over these domains.

Name	Notation	Description
Data	\mathbf{D}	domain of the given data structure
	$d \in \mathbf{D}^*$	sequences form the data domain
Data operations	$o \in \mathbf{O}^{\mathbf{D}}$	operations of the given data structure
Symbols	$s \in \$y$	for naming values
Forms	$\varphi \in \mathbf{F}$	expressions describing computations
Continuation forms	$\mathbf{F}_{cs} \subset \mathbf{F}$	for making continuation segments
Environments	$\xi \in \mathbf{E}$	finite maps from symbols to values
Dtrees	$\delta \in \mathbf{D}_t$	description trees
Pfns	$p \in \mathbf{P}$	descriptions of partial functions
Continuations	$\gamma \in \mathbf{C}_o$	descriptions of computation contexts
Operations	$\vartheta \in \mathbf{O}$	data operations, pfns, continuations
Computation Domain	$a \in \mathbf{V}$	data and operations
Values	$u, v \in \mathbf{V}^*$	sequences from computation domain
States	$\zeta \in \$t$	states of sequential computations

Figure 2. Rum Domains

As discussed in §3 the *Rum* domains are essentially many-sorted initial algebras. They are presented by the defining equations given in Figure 3. The summands in the equations indicate the name and arity of the domain element constructors. A constructor applied to a tuple of domains denotes the set of objects obtained by applying the constructor to all tuples of objects in the tuple of domains. For example λ is a form constructor with two arguments, a symbol and a form and $\lambda(\mathbf{S}_y, \mathbf{F})$ is the set of forms $\lambda(s, \varphi)$ for s a symbol and φ a form. The defined domains are the least sets satisfying these equations. Elements with different constructions are distinct unless they can be proved equal by the equality rules to be explained below.

$$\begin{aligned}
\mathbf{IF} &\sim \mathbf{S}_y \oplus \lambda(\mathbf{S}_y, \mathbf{F}) \oplus \mathbf{app}(\mathbf{F}, \mathbf{F}) \oplus \mathbf{if}(\mathbf{F}, \mathbf{F}, \mathbf{F}) \oplus \mathbf{mt} \oplus \mathbf{cart}(\mathbf{F}, \mathbf{F}) \oplus \mathbf{fst}(\mathbf{F}) \oplus \mathbf{rst}(\mathbf{F}) \\
&\quad \oplus \mathbf{top} \oplus \mathbf{note}(\mathbf{S}_y, \mathbf{F}) \\
\mathbf{F}_{cs} &\sim \mathbf{ifi}(\mathbf{F}, \mathbf{F}) \oplus \mathbf{appi}(\mathbf{F}) \oplus \mathbf{appc}(\mathbf{S}_y) \oplus \mathbf{carti}(\mathbf{F}) \oplus \mathbf{cartc}(\mathbf{S}_y) \oplus \mathbf{fstc} \oplus \mathbf{rstc} \\
\mathbf{Dt} &\sim \{\langle \varphi : \xi \rangle \mid \mathbf{Frees}(\varphi) \subset \mathbf{Dom}(\xi)\} \\
\mathbf{P} &\sim \{\langle \lambda(s, \varphi) : \xi \rangle \mid \mathbf{Frees}(\lambda(s, \varphi)) \subset \mathbf{Dom}(\xi)\} \\
\mathbf{Co} &\sim \mathbf{top} \oplus \{\mathbf{Co} \circ \langle \varphi : \xi \rangle \mid \mathbf{Frees}(\varphi) \subset \mathbf{Dom}(\xi) \wedge \varphi \in \mathbf{F}_{cs}\} \\
\mathbf{S}_t &\sim (\mathbf{Co} \triangleright \mathbf{Dt}) \oplus (\mathbf{Co} \triangleleft \mathbf{V}^*) \\
\mathbf{O} &\sim \mathbf{O}^D \oplus \mathbf{P} \oplus \mathbf{Co} \quad \mathbf{V} \sim \mathbf{D} \oplus \mathbf{O} \quad \mathbf{E} \sim [\mathbf{S}_y \xrightarrow{t} v']
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{ifi}(\varphi_1, \varphi_2) &= \lambda(s_*, \mathbf{if}(s_*, \varphi_1, \varphi_2)) \\
\mathbf{appi}(\varphi_1) &= \lambda(s_*, \mathbf{app}(s_*, \varphi_1)) \\
\mathbf{appc}(s) &= \lambda(s_*, \mathbf{app}(s, s_*)) \\
\mathbf{carti}(\varphi_1) &= \lambda(s_*, \mathbf{cart}(s_*, \varphi_1)) \\
\mathbf{cartc}(s) &= \lambda(s_*, \mathbf{cart}(s, s_*)) \\
\mathbf{fstc} &= \lambda(s_*, \mathbf{fst}(s_*)) \\
\mathbf{rstc} &= \lambda(s_*, \mathbf{rst}(s_*))
\end{aligned}$$

with s_* chosen so as to occur only where explicitly shown.

Figure 3. Domain equations

About Forms. Expressions called *forms* are the syntactic objects – our programming language. Forms are generated from the set $\$y$ of symbols and the constants `mt` and `top` (for naming the empty sequence and the empty context respectively)¹ by the constructions `app` (application), λ (function abstraction), `note` (control abstraction), if (conditional), `cart` (sequence concatenation), and `fst` and `rst` (sequence selection). `note` is the *Rum* analog of **Landin’s J** operator, and the *catch* construct of Scheme [Steele and Sussman 1975] (known as `call/cc` in modern versions of Scheme [Rees and Clinger 1986]). It binds the continuation representing the current computation context to the noted symbol. For conditional branching, the empty sequence represents false and any non-empty sequence represents true. λ and `note` are binding constructs. Bound and free occurrences of symbols in forms are defined as usual and $\text{Free}(φ)$ is the set of symbols occurring free in $φ$. Two forms are equal if they have the same construction modulo renaming of bound variables (a-conversion). $φ\{s/φ_0\}$ is the result of replacing free occurrences of s in $φ$ by $φ_0$ (with renaming of bound symbols in $φ$ if necessary to avoid trapping of free symbols in $φ_0$). We will use c, f, g, h, x, y, z and other identifiers not otherwise reserved to denote particular symbols when writing particular forms. Distinct identifiers are assumed to denote distinct symbols. We use traditional notation for application and abstraction. Thus we generally write $\lambda x.f(x)$ for $\lambda(f, \text{app}(f, x))$. Other convenient abbreviations and notation will be given later.

About Dtrees. To describe a particular computation, a form is closed in an *environment* that assigns a sequence of elements from the computation domain to the free symbols. This closure is called a *dtree* (for description tree) to emphasize the relation between the tree structure of a form and the local structure of the computation described by the form. Two dtrees are equal just when they differ by renaming of environment bound symbols (i); by renaming of λ - or `note`- bound symbols (ii); or by modifying bindings of symbols not free in the form (iii).

- (i) $(\text{cart}(x, x) : ξ) = (\text{cart}(x, y) : ξ) \quad ;; \text{ if } ξ(x) = ξ(y)$
- (ii) $(\lambda x.\text{note}(c)f(\text{cart}(c, x)) : ξ) = (\lambda y.\text{note}(k)f(\text{cart}(k, y)) : ξ)$
- (iii) $(x : ξ) = (x : ξ\{y ← v\})$

Note that by our convention in (iii) x and y are distinct symbols.

About values. The computation domain contains data, data operations, *pfn*s, and *continuations*.

¹ We use `top` to denote both the form and the corresponding continuation. Context will determine the meaning in cases where the distinction is important.

Structurally pfns are just X-dtrees and two pfns are equal just when the corresponding dtrees are equal. Computationally pfns are functional abstractions analogous to Landin's closures. The computation described by application of a pfn to a value is that described by the dtree whose form is the body of the lambda form and whose environment is the result of binding the value to the lambda variable in the pfn environment.

Continuations represent computation contexts built up in the process of carrying out computations. Structurally they are sequences of dtrees whose form components are continuation segment forms. Two continuation segments are equal just when the corresponding dtrees are equal and two continuations are equal just when they have the same length and corresponding segments are equal. Computationally continuations are control abstractions. Applying a continuation to a value means resuming computation in the context represented by the continuation, returning the value and discarding the current context.

The only further rules of equality are the equality rules for finite maps and finite sequences explained in §3.

4.2. Operations and relations.

Figure 4 summarizes the operations and relations on *Rum* domains defined below. The sign for each operation or-relation is given together with its arity.

Name	Notation and arity
Single Step	$\rightarrow_s \in [\mathbb{S} \xrightarrow{P} \mathbb{S}]$
Step	$\rightarrow_c \in [\mathbb{S} \times \mathbb{S}]$
C-seq formation	$C_s \in [\mathbb{D} \xrightarrow{P} \mathbb{S}^*]$
Evaluation	$\hookrightarrow \in [\mathbb{D} \xrightarrow{P} \mathbb{V}^*]$
Reduces-to	$\rightarrow_c \in [\mathbb{D} \times \mathbb{D}]$
Definedness	$\Downarrow \subset \mathbb{D}$

Figure 4. Relations and operations of *Rum*

4.2.1. Computation rules.

Computation is modeled as a process of generating sequences of structures called *computation states*. This is similar to the approach of Wegner. Computation states together with the *single step* (next state) relation constitute an abstract machine analogous to the SECD machine of Landin. The *Rum* abstract machine is very similar to the abstract machine implicit in the first-order continuation passing interpreter of Reynolds and to the abstract machine from which Felleisen and Friedman derive their control calculus.

Computation states come in two flavors begin **states** $\gamma \triangleright \delta$ constructed from a continuation γ and a dtree δ and return states $\gamma \triangleleft v$ constructed from a continuation γ and a value v . The continuation component of a state represents the current context. The dtree component of a begin state represents the current expression – the subcomputation to begin next. A return state corresponds to returning the value component to the current context. Continuations are compositions of segments corresponding to steps in building up the context represented. The continuation **top** represents the empty context (top level). If γ represents the context for $\langle \varphi_0(\varphi_1) : \xi \rangle$, the application of φ_0 to φ_1 in environment ξ , then $\gamma \circ \langle \text{appi}(\varphi_1) : \xi \rangle$ represents the context for the evaluation of the function part $\langle \varphi_0 : \xi \rangle$. If $\langle \varphi_0 : \xi \rangle$ returns value v then $\gamma \circ \text{appc}(v)$ represents the context for evaluation of the argument part $\langle \varphi_1 : \xi \rangle$. If γ represents the context for the evaluation of $\langle \text{if}(\varphi_0, \varphi_1, \varphi_2) : \xi \rangle$ then $\gamma \circ \langle \text{ifi}(\varphi_1, \varphi_2) : \xi \rangle$ represents the context for the evaluation of the test part $\langle \varphi_0 : \xi \rangle$. Similarly for **cart** (segment constructors **carti**, **cartc**) **fst** (segment constructor **fstc**), and **rst** (segment constructor **rstc**). Note that we use some abbreviations in writing continuation segments. For example **appc**(v) abbreviates $\langle \text{appc}(x) : x \leftarrow v \rangle$ since the choice of symbol is irrelevant, and **fstc** abbreviates $\langle \text{fstc} : \xi \rangle$ since the environment is irrelevant.

Computation proceeds by applying the step rules to obtain a sequence of **states** called a computation sequence. The form and continuation segment constructors correspond to abstract machine instructions. For begin states $\gamma \triangleright \delta$ the rule to be applied is determined by the construction of the form component of 6. If δ is primitive its value is returned to γ . Otherwise a subexpression of δ is selected, a continuation segment is added to the continuation recording the information needed to continue when the value of the selected subexpression is returned, and computation described by the subexpression is begun. A return state **top** $\triangleleft v$ is terminal. For return states $(\gamma \circ \psi) \triangleleft v$ the rule to be applied is determined by the component of the top segment ψ . If this corresponds to a basic computation primitive (**ifi**, **appc**, **cartc**, **fstc**, **rstc**) it is executed. Otherwise the next subexpression to process is chosen and the top segment is replaced by a new segment that remembers the returned value and any additional information needed.

\rightarrow_i is the least relation on states satisfying the following:

- (sym) $\gamma \triangleright \langle s : \xi \rangle \rightarrow_i, \gamma \triangle \xi(s)$
- (lam) $\gamma \triangleright \langle \lambda(s, \varphi_{\text{body}}) : \xi \rangle \rightarrow_i, \gamma \triangle \langle \lambda(s, \varphi_{\text{body}}) : \xi \rangle$
- (app) $\gamma \triangleright \langle \text{app}(\varphi_{\text{fun}}, \varphi_{\text{arg}}) : \xi \rangle \rightarrow_i, \gamma \circ \langle \text{appi}(\varphi_{\text{arg}}) : \xi \rangle \triangleright \langle \varphi_{\text{fun}} : \xi \rangle$
- (appi) $\gamma \circ \langle \text{appi}(\varphi_{\text{arg}}) : \xi \rangle \triangle v_{\text{fun}} \rightarrow_i, (\gamma \circ \text{appc}(v_{\text{fun}})) \triangleright ((\text{Pug} : \xi))$
- (o) $\gamma \circ \text{appc}(o) \triangle d \rightarrow_i, \gamma \triangle o(d)$
- (appc) $\gamma \circ \text{appc}(\langle \lambda(s, \varphi) : \xi \rangle) \triangle v_{\text{arg}} \rightarrow_i, \gamma \triangleright \langle \varphi : \xi \{s \leftarrow v_{\text{arg}}\} \rangle$
- (SW) $\gamma \circ \text{appc}(\gamma_0) \triangle v_{\text{arg}} \rightarrow_i, \gamma_0 \triangle v_{\text{arg}}$
- (if) $\gamma \triangleright \langle \text{if}(\varphi_{\text{test}}, \varphi_{\text{then}}, \varphi_{\text{else}}) : \xi \rangle \rightarrow_i, \gamma \circ \langle \text{ifi}(\varphi_{\text{then}}, \varphi_{\text{else}}) : \xi \rangle \triangleright \langle \varphi_{\text{test}} : \xi \rangle$
- (ifi) $\gamma \circ \langle \text{ifi}(\varphi_{\text{then}}, \varphi_{\text{else}}) : \xi \rangle \triangle v_{\text{test}} \rightarrow_i, \gamma \triangleright \begin{cases} \langle \varphi_{\text{then}} : \xi \rangle & \text{if } v_{\text{test}} \neq \square \\ \langle \varphi_{\text{else}} : \xi \rangle & \text{if } v_{\text{test}} = \square \end{cases}$
- (mt) $\gamma \triangleright \text{mt} \rightarrow_i, \gamma \triangle \square$
- (Cart) $\gamma \triangleright \langle \text{cart}(\varphi_{\text{lhs}}, \varphi_{\text{rhs}}) : \xi \rangle \rightarrow_i, \gamma \circ \langle \text{carti}(\varphi_{\text{rhs}}) : \xi \rangle \triangleright \langle \varphi_{\text{lhs}} : \xi \rangle$
- (carti) $\gamma \circ \langle \text{carti}(\varphi_{\text{rhs}}) : \xi \rangle \triangle v_{\text{lhs}} \rightarrow_i, \gamma \circ \text{cartc}(v_{\text{rhs}}) \triangleright \langle \varphi_{\text{rhs}} : \xi \rangle$
- (cartc) $\gamma \circ \text{cartc}(v_{\text{lhs}}) \triangle v_{\text{rhs}} \rightarrow_i, \gamma \triangle [v_{\text{lhs}}, v_{\text{rhs}}]$
- (fst) $\gamma \triangleright \langle \text{fst}(\varphi_{\text{seq}}) : \xi \rangle \rightarrow_i, \gamma \circ \text{fstc} \triangleright \langle \varphi_{\text{seq}} : \xi \rangle$
- (fstc) $\gamma \circ \text{fstc} \triangle v_{\text{seq}} \rightarrow_i, \gamma \triangle 1^{\text{st}}(v_{\text{seq}})$
- (rst) $\gamma \triangleright \langle \text{rst}(\varphi_{\text{seq}}) : \xi \rangle \rightarrow_i, \gamma \circ \text{rstc} \triangleright \langle \varphi_{\text{seq}} : \xi \rangle$
- (rstc) $\gamma \circ \text{rstc} \triangle v_{\text{seq}} \rightarrow_i, \gamma \triangle r^{\text{st}}(v_{\text{seq}})$
- (top) $\gamma \triangleright \text{top} \rightarrow_i, \gamma \triangle \text{top}$
- (note) $\gamma \triangleright \langle \text{note}(s)\varphi : \xi \rangle \rightarrow_i, \gamma \triangleright \langle \varphi : \xi \{s \leftarrow \gamma\} \rangle$

\rightarrow is the transitive reflexive closure of \rightarrow_i .

Figure 5. Rules for stepping

4.2.2. Computation sequences.

□ **(Steps):** The single-step relation (\rightarrow_i) and the step relation (\rightarrow) are defined in Figure 5. A step is a pair of states (ζ_0, ζ_1) such that $\zeta_0 \rightarrow_i \zeta_1$.²

It is easy to see from the definition that the single-step relation is a functional relation. This is expressed by the unicity lemma.

Lemma (single-step is functional): $\zeta \rightarrow_i \zeta_0 \wedge \zeta \rightarrow_i \zeta_1 \Rightarrow \zeta_0 = \zeta_1$.

² One might also include the rule applied as part of a step. In Rum the rule is uniquely determined by the first state.

$$\begin{aligned}
 & \gamma \triangleright (\text{if}(\text{Zerop}(x), c(x), x) : \xi) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle \triangleright (\text{Zerop}(x) : \xi) && (\text{if}) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle 0 \langle \text{appi}(x) : \xi \rangle \triangleright (\text{Zerop} : \xi) && (\text{app}) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle 0 \langle \text{appi}(x) : \xi \rangle \Delta \text{Zerop} && (\text{sym}) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle \circ \text{appc}(\text{Zerop}) \triangleright (x : \xi) && (\text{appi}) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle 0 \text{ appc}(\text{Zerop}) \Delta 0 && (\text{sym}) \\
 \rightsquigarrow & \gamma 0 \langle \text{ifi}(c(x), x) : \xi \rangle \Delta 0 && (\text{Zerop}) \\
 \rightsquigarrow & \gamma \triangleright (c(x) : \xi) && (\text{ifi}) \\
 \rightsquigarrow & \gamma 0 \langle \text{appi}(x) : \xi \rangle \triangleright (c : \xi) && (\text{app}) \\
 \rightsquigarrow & \gamma 0 \langle \text{appi}(x) : \xi \rangle \Delta \text{top} && (\text{sym}) \\
 \rightsquigarrow & \gamma 0 \text{ appc}(\text{top}) \triangleright (x : \xi) && (\text{appi}) \\
 \rightsquigarrow & \gamma 0 \text{ appc}(\text{top}) \Delta 0 && (\text{sym}) \\
 \rightsquigarrow & \text{top} \Delta 0 && (\text{sw})
 \end{aligned}$$

where ξ maps x to 0 , c to top , and
maps the symbol Zerop to the S-expression operation Zerop .

Figure 6. Example computation sequence

▷ **(Computation sequences.):** A non-empty (possibly infinite) sequence of states Σ is a computation sequence if each adjacent pair of states is a single step – $(\forall i < |\Sigma| - 1)(\Sigma \downarrow_i \rightsquigarrow_{\text{t}} \Sigma \downarrow_{i+1})$. We write $\zeta_a \xrightarrow{\Sigma} \zeta_z$ if Σ is a finite computation sequence of length $n + 1$ such that $\Sigma \downarrow_0 = \zeta_a$ and $\Sigma \downarrow_n = \zeta_z$. $Cs(\zeta)$ is the longest computation sequence with first element ζ . By the functionality of single step there is a unique (possibly infinite) such sequence. $Cs(\delta)$, the computation sequence for the dtree 6, is defined as $Cs(\text{top} \triangleright \delta)$.

To illustrate the structure and properties of computation we carry out in detail the computation described by the form $\text{if}(\text{Zerop}(x), c(x), x)$ closed in an environment ξ mapping x to 0 and c to top . The computation is carried out in an arbitrary context γ . The resulting computation sequence, Σ_{ex} , is shown in Figure 6. The rule applied for each step is given in the column at the right.

4.3. Abbreviations and notation.

We close this section with some syntactic sugar. We use two kinds of definitions – syntactic abbreviations (macros) and definitions of “constants”. The sign \coloneqq is used for syntactic abbreviations (macro expansion) and the sign \leftarrow is used for constant definitions. Defining a constant has the effect of restricting environments to map the constant symbol to its value.

D (application macros): Multi-ary application and abstraction can be expressed by currying or by using sequences and let abbreviates lambda application as usual.

$$\begin{aligned}\lambda s_1, \dots, s_n. \varphi &\coloneqq \lambda(s_1, \dots, \lambda(s_n, \varphi) \dots), \\ \varphi_0(\varphi_1, \dots, \varphi_n) &\coloneqq \mathbf{app}(\dots \mathbf{app}(\varphi_0, \varphi_1), \dots \varphi_n), \\ [\varphi_1, \dots, \varphi_n] &\coloneqq \mathbf{cart}(\varphi_1, \mathbf{cart}(\dots, \varphi_n)), \\ \lambda[s_1, s_2, \dots, s_n]\varphi &\coloneqq \lambda s_1.((\lambda s_1. \lambda[s_2, \dots, s_n]\varphi)(\mathbf{fst}(s_1), \mathbf{rst}(s_1))), \quad \text{and} \\ \mathbf{let}\{b \leftarrow \varphi\}\varphi_{\mathbf{body}} &\coloneqq (\lambda b. \varphi_{\mathbf{body}})(\varphi).\end{aligned}$$

where b is a symbol or a sequence of symbols.

We define the constants I (the identity pfn), B (the composition pfn), and **Rec** (the recursion pfn) as follows.

$$\begin{aligned}\mathsf{D} \quad \mathsf{I} &\leftarrow \lambda x. x \\ \mathsf{D} \quad \mathsf{B} &\leftarrow \lambda f, g, x. f(g(x)) \\ \mathsf{D} \quad \mathsf{Rec} &\leftarrow \lambda g. \mathbf{let}\{h \leftarrow \lambda h. \lambda x. g(h(h), x)\} h(h)\end{aligned}$$

We often use infix notation for “binary” pfns. For example $p_0 \circ p_1$ is the composition $B(p_0, p_1)$

Recursive definitions are represented using the recursion pfn. A recursion equation of the form

$$f(b_1, \dots, b_n) \leftarrow \varphi$$

where b_i is a symbol or a sequence of symbols, is an abbreviation of

$$f \leftarrow \mathbf{Rec}(\lambda f. \lambda b_1, \dots, b_n. \varphi).$$

For example

$$\mathsf{D} \quad \mathsf{Bot} \leftarrow \lambda x. \mathsf{Bot}(x)$$

defines **Bot** to be $\text{Rec}(\lambda f.\lambda x.f(x))$ ($= \text{Rec}(\lambda \text{Bot}.\lambda x.\text{Bot}(x))$). Thus **Bot** is the totally undefined (bottom) pfn.³

▫ **(boolean macros):** The boolean operations **and**, **or**, and **not** are defined using **if** following the Lisp tradition. Here **T** is any expression with non-empty value - for example **I**.

$$\begin{aligned}\text{or}(\varphi_{\text{lhs}}, \varphi_{\text{rhs}}) &:= \text{if}(\varphi_{\text{lhs}}, \text{T}, \text{if}(\varphi_{\text{rhs}}, \text{T}, \text{mt})) \\ \text{and}(\varphi_{\text{lhs}}, \varphi_{\text{rhs}}) &:= \text{if}(\varphi_{\text{lhs}}, \text{if}(\varphi_{\text{rhs}}, \text{T}, \text{mt}), \text{mt}) \\ \text{not}(\varphi) &:= \text{if}(\varphi, \text{mt}, \text{T})\end{aligned}$$

4.4. Operational relations

We call properties of pfns that depend only on their “black box” behavior extensional properties. Such properties do not depend on the computation described but only on the results of application. Many of the properties we wish to prove about programs are extensional properties. For example

- **Tpc**, and **Tps** compute the same function as **Tp**.
- **C32** converts a sequence of characters segmented as strings of length three into the same sequence of characters segmented as strings of length two.
- **Sieve** generates the sequence of all prime numbers in increasing order

The *operational approximation and equivalence* relations on dtrees and values formalize the notion of “same black box behavior”. The basic idea is that two objects are equivalent if they can not be distinguished by any computational context (continuation). Operational equivalence in *Rum* is defined in terms of the abstract machine described in §4. It is an adaptation of [Plotkin 1975], which is in turn an adaptation of the notion of extensional equivalence [Morris 1968]. Operational approximation and equivalence are preserved by substitution and abstraction and thus provide a good basis for equational reasoning. For most pfns, the intended domain of application is some subset U of V^* and we are really interested in proving statements of the form “for u ranging over $U \dots u \dots$ ”. For this reason, we also define the *operational membership* relation on dtrees and subsets of values. Operational membership in some set of values expresses the fact that a dtree can be treated as if it were a value in that set.⁴

³ We consider the case of a pfn defined by a single equation. The multiple recursion case is similar, just messier.

⁴ The traditional formulation of operational equivalence (cf. [Plotkin 1975]) is between forms (expressions) rather than dtrees (closures). In the appendix we show that our formulation when extended to forms is equivalent to the standard formulation.

We begin by defining the trivial approximation relation \sqsubseteq_0 on values which relates values of the same length whose corresponding elements are either equal data elements or both operations (but not necessarily equal). This is our version of indistinguishable. Two states are trivially approximate if the first steps to a value implies that both step to related values.

▷ (**Trivial approximation**):

$$\begin{aligned} v_0 \sqsubseteq_0 v_1 &\stackrel{\text{df}}{\equiv} |v_0| = |v_1| \wedge (\forall i < |v_0|)(v_0 \downarrow_i \in \mathbf{D} \vee v_1 \downarrow_i \in \mathbf{D} \Rightarrow v_0 \downarrow_i = v_1 \downarrow_i) \\ \zeta_0 \sqsubseteq_0 \zeta_1 &\stackrel{\text{df}}{\equiv} (\forall v_0)(\zeta_0 \rightarrowtail \text{top} \triangle v_0 \Rightarrow (\exists v_1)(v_0 \sqsubseteq_0 v_1 \wedge \zeta_1 \rightarrowtail \text{top} \triangle v_1)) \end{aligned}$$

Operational approximation \sqsubseteq is defined in terms of trivial approximation by saying that two dtrees or values are operationally approximate just if for any continuation the corresponding states are trivially approximate. Operational equivalence (\cong) is the intersection of operational approximation with its inverse. Operational membership $\tilde{\in}$ in a set of values means operational equivalence to a value in that set.

▷ (**Operational relations**):

$$\begin{aligned} \delta_0 \sqsubseteq \delta_1 &\stackrel{\text{df}}{\equiv} (\forall \gamma)(\gamma \triangleright \delta_0 \sqsubseteq_0 \gamma \triangleright \delta_1) \\ v_0 \sqsubseteq v_1 &\stackrel{\text{df}}{\equiv} (\forall \gamma)(\gamma \triangle v_0 \sqsubseteq_0 \gamma \triangle v_1) \\ \chi_0 \cong \chi_1 &\stackrel{\text{df}}{\equiv} \chi_0 \sqsubseteq \chi_1 \wedge \chi_1 \sqsubseteq \chi_0 \quad :: \chi \text{ a dtree or value} \\ v \tilde{\in} U &\stackrel{\text{df}}{\equiv} (\exists u \in U)(v \cong u) \quad :: U \text{ a subset of } \mathbf{V}^* \end{aligned}$$

The operational relations are extended to relate dtrees and values by defining $\delta R v \stackrel{\text{df}}{\equiv} \delta R \langle x : x \dashv v \rangle$ for R one of \sqsubseteq, \cong and $\delta \tilde{\in} U \stackrel{\text{df}}{\equiv} (\exists u \in U)(\delta \cong u)$ for U a subset of \mathbf{V}^* . Operational membership in the set \mathbf{V}^* expresses definedness in the strong sense that a value is returned to any calling context.

Some key properties of operational approximation and equivalence are stated below. These are all the properties that are needed to develop our first-order theory. Proofs can be found in the appendix.

Theorem (Pre-order and equivalence): \sqsubseteq and \cong are reflexive transitive relations and \cong is symmetric (hence indeed an equivalence).

Theorem (Pointwise): Approximate values are pointwise approximate.

$$v_0 \sqsubseteq v_1 \Rightarrow |v_0| = |v_1| \wedge (\forall i < |v_0|)(v_0 \downarrow_i \sqsubseteq v_1 \downarrow_i)$$

Corollary (Pointwise): Data values are approximate only if equal.

$$d_0 \sqsubseteq d_1 \Leftrightarrow d_0 = d_1$$

Theorem (Computational facts):

- (i) $\zeta_0 \succcurlyeq \zeta_1 \Rightarrow \text{co co } \zeta_1 \wedge \text{cl} \sqsubseteq_0 \zeta_0$
- (ii) $\text{co } \succcurlyeq \text{top} \Delta v_0 \wedge \zeta_1 \succcurlyeq \text{top} \Delta v_1 \Rightarrow \zeta_0 \sqsubseteq_0 \zeta_1 \Leftrightarrow v_0 \sqsubseteq_0 v_1$

To formalize the notion of substitution into semantic (dtrees, values, continuations, and states) entities we consider entities with (zero or more) holes for dtrees, continuations, or values. We will only need to consider entities with a single sort of hole. Such entities are built in the same way as ordinary entities except that we add a clause asserting that holes of a given sort are objects of that sort. Then $\chi[\delta]$ is the result of filling the hole with 6. Similarly for other sorts of holes.

Theorem (Substitution): For χ a dtree or value

- (subst.c) $\gamma_0 \sqsubseteq \gamma_1 \Rightarrow \chi[\gamma_0] \sqsubseteq \chi[\gamma_1]$
- (subst.dt) $\delta_0 \sqsubseteq \delta_1 \Rightarrow \chi[\delta_0] \sqsubseteq \chi[\delta_1]$
- (subst.v) $v_0 \sqsubseteq v_1 \Rightarrow \chi[v_0] \sqsubseteq \chi[v_1]$

Corollary (subst.env):

$$(\forall s \in \text{Frees}(\varphi)(\xi_0(s) \sqsubseteq \xi_1(s)) \Rightarrow \langle \varphi : \xi_0 \rangle \sqsubseteq \langle \varphi : \xi_1 \rangle)$$

For full generality we consider dtrees as being generated by constructors analogous to forms. Thus dtree holes can occupy the position of a free variable in a form. This leads to the following corollary.

Corollary (subst.exp): If $\delta_0 = \langle \varphi_0 : \xi \rangle \sqsubseteq \delta_1 = \langle \varphi_1 : \xi \rangle$ and $\delta[\] = \langle \varphi\{x/\]\} : \xi \rangle$ then

$$\delta[\delta_0] = \langle \varphi\{x/\varphi_0\} : \xi \rangle \sqsubseteq \delta[\delta_1] = \langle \varphi\{x/\varphi_1\} : \xi \rangle.$$

Theorem (Extensionality):

- (ext.op) $\vartheta_0 \sqsubseteq \vartheta_1 \Leftrightarrow (\forall v)(\vartheta_0(v) \sqsubseteq \vartheta_1(v))$
- (ext.co) $\gamma_0 \sqsubseteq \gamma_1 \Leftrightarrow (\forall v)(\gamma_0 \Delta v \sqsubseteq \gamma_1 \Delta v)$

where by abuse of notation we write $\vartheta(v)$ for $\langle \text{app}(f, x) : f \leftarrow \mathbf{6}, x \leftarrow v \rangle$, etc.

The recursion theorem says that the recursion pfn Rec computes the least fixedpoint of pfns — closures of forms $\lambda f.\lambda x.\varphi$.

Theorem (Recursion): Let ϑ be a closure of $\lambda f.\lambda x.\varphi$ then the recursion operator Rec computes the least fixed-point of ϑ with respect to the operational approximation ordering.

$$(\text{fix}) \quad \text{Rec}(\vartheta) \cong \vartheta(\text{Rec}(\vartheta))$$

$$(\text{min}) \quad \vartheta(\vartheta_0) \sqsubseteq \vartheta_0 \Rightarrow \text{Rec}(\vartheta) \sqsubseteq \vartheta_0$$

5. Towards a theory of function and control abstractions

The operational approximation and equivalence allow one to forget the details of how a computation is carried out, but one must still use dtrees and states as well as values in reasoning about properties of pfns. We would like to work only with values. To do this we define a language for expressing extensional properties of elements of the computation domain. In this language forms play the role of terms and atomic formulas are based on the operational approximation and membership relations. The semantics of the language is based on notions of satisfaction and truth in the *Rum* model using environments as interpretations. The formal theory is **parameterized** by a data structure \mathfrak{D} just as the underlying intensional semantics is. The axioms and rules fall into three groups. The basic logical theory includes the first-order logic of partial **terms**, laws for defining and proving membership in sets of values (“in laws”), and induction schemes. The computational laws include the theory of the underlying data structure, rules for reasoning about sequences, and rules simulating reduction steps. The **final** group of laws concern extensionality and the recursion theorem. We derive **some** additional laws to illustrate the use of the basic laws and to develop tools for proving properties of particular pfns.

5.1. The language and its semantics

We now develop a language. and theory for expressing and proving extensional properties of elements of the computation domain.

▷ **(Terms):** The terms of the language are just forms where we partition symbols into variables and constants. For variable symbols we will use italic identifiers such as x , f , out , etc. For constant symbols we will use identifiers in **ThisFont**. The constant symbols will be used to give names to data elements, data operations, pfns and other defined constants and will generally depend on the particular choice of data structure \mathfrak{D} . We will not distinguish between constant symbols and the constants they denote. For example in the case of S-expressions the symbol **PairMk** is a constant symbol denoting the S-expression operation **PairMk**. In all cases **Rec** is a constant symbol denoting the recursion pfn **Rec**.

▷ **(Atomic formulae):** Atomic formulae are $\varphi_0 \sqsubseteq \varphi_1$ and $\varphi_0 \in U$ for each subset U of \mathbf{V}^* .¹ We will use U, U_0, \dots as metavariables ranging over subsets of \mathbf{V}^* .

▷ **(Formulae):** Formulae are built from atomic formulae in the usual way using the logical connectives and quantifiers (A, \neg, V). The free and bound variables of a formula are determined as usual. We will use Φ, Φ_0, \dots to denote formulae.

¹ In practice we use only subsets **definable by simple** operations.

The meaning of a formula is defined by saying when it is true, which in turn is defined in terms of the notion of satisfaction. Intuitively an environment ξ satisfies a formula Φ iff the closure of Φ is true of the corresponding dtrees. In a statement of the form $\xi \models \Phi$ we assume that the domain of ξ contains the free variables of Φ , and that constant symbols are interpreted by their **defined** values. The satisfaction relation is **defined** by induction on the construction of formulae. The key is for the atomic case. The remaining cases are the usual (**Tarskian**) definition.

▷ **(Satisfaction):**

$$\begin{aligned}\xi \models \varphi_0 \sqsubseteq \varphi_1 &\Leftrightarrow (\varphi_0 : \xi) \sqsubseteq (\varphi_1 : \xi) \\ \xi \models \varphi \in U &\Leftrightarrow (\varphi : \xi) \in U \\ \xi \models \Phi_0 \wedge \Phi_1 &\Leftrightarrow (\xi \models \Phi_0 \wedge \xi \models \Phi_1) \\ \xi \models \neg\Phi_0 &\Leftrightarrow \neg(\xi \models \Phi_0) \\ \xi \models (\forall x)\Phi &\Leftrightarrow (\forall v)(\xi\{x \mapsto v\} \models \Phi)\end{aligned}$$

▷ **(Truth):** A formula is true if all environments (that map constant symbols are mapped to their defined values) satisfy it.

$$\models \Phi \Leftrightarrow (\forall \xi)(\xi \models \Phi)$$

We will generally omit the \models sign and simply write Φ to assert that Φ is true.

▷ **(Abbreviations):** We will treat \sqsupseteq , \cong , and the logical connectives (\vee , \Rightarrow , \Leftrightarrow , \exists) as syntactic abbreviations in the usual manner. Thus we have

$$\begin{aligned}\varphi_0 \sqsupseteq \varphi_1 &:= \varphi_1 \sqsubseteq \varphi_0 \\ \varphi_0 \cong \varphi_1 &:= \varphi_0 \sqsubseteq \varphi_1 \wedge \varphi_1 \sqsubseteq \varphi_0 \\ \Phi_0 \vee \Phi_1 &:= \neg(\neg\Phi_0 \wedge \neg\Phi_1) \\ \Phi_0 \Rightarrow \Phi_1 &:= \neg\Phi_0 \vee \Phi_1 \\ \Phi_0 \Leftrightarrow \Phi_1 &:= \Phi_0 \Rightarrow \Phi_1 \wedge \Phi_1 \Rightarrow \Phi_0 \\ (\exists x)\Phi &:= \neg(\forall x)(\neg\Phi_0)\end{aligned}$$

We often write formulas using bounded quantifiers. This is an abbreviation in the usual manner. Thus $(\forall u \in U)\Phi$ abbreviates $(\forall u)(u \in U \Rightarrow \Phi)$ and $(\exists u \in U)\Phi$ abbreviates $(\exists u)(u \in U \wedge \Phi)$.

D (Sorts): A sort is a non-empty subset of \mathbf{V}^* .² Some examples are \mathbf{V}^* , \mathbf{V} , \mathbf{O} , \mathbf{P} , and \mathbf{Co} . If U is a sort we may declare (locally) that some variable u ranges over U in expressing properties and giving proofs. Thus we **may** assert “for u ranging over U we have Φ ”. This is equivalent to the assertion “ $u \in U \Rightarrow \Phi(u)$ ”. For example working in the S-expression context if we specify z ranges over integers and c ranges over continuations then $\text{Cons}[z, c(y)] \cong c(y)$ is equivalent to

$$z \in \mathbf{D}_{\text{int}} \wedge c \in \mathbf{Co} \Rightarrow \text{Cons}[z, c(y)] \cong c(y)$$

and means that

$$\{z \in i, c \in \gamma, y \in v\} \models \text{Cons}[z, c(y)] \cong c(y)$$

for all integers i , continuations γ and values v .

A variable is assumed to range over \mathbf{V}^* unless otherwise declared. We may write $(\forall x \in \mathbf{V}^*)$ to emphasize that x is required to range over all values. The sort of a variable occurring bound in a form has no formal significance. Sorted variables are used in such contexts just to indicate the intended range.

5.2. The basic theory

We observe two conventions in presenting the laws of our theory: free variables are implicitly universally quantified; and if metavariables ranging over forms occur in a law then it is a schema representing a family of laws one for each assignment of forms to the form variables $\varphi, \varphi_0, \dots$. We use the sign \blacksquare to indicate axioms and rules of our theory. The fact that they are true according to our definition of satisfaction means that any derived consequences will also be true in our model (or any other model of the axioms and rules).

5.2.1. Logic of partial terms and approximation

It is clear from the definitions of truth in Rum that we have ordinary classical truth relative to the atomic propositions about operational approximation and membership. The only trick is that we must account for the fact that for a given interpretation of free variables a form may not have a well defined denotation – the computation described may be context-dependent or may diverge. Thus the logic is a classical logic of partial terms. This means that eliminating (instantiating) universal quantifiers must be restricted to terms that are operationally equivalent to values. The point is that in order to permit instantiation of a universal variable

² Non-emptiness is needed to preserve the classical interpretation of quantifiers relativized to sorts. See for example [Goguen and Meseguer 1984] for discussion of problems that arise when variables are allowed to range over empty domains.

by φ we must insure that q is operationally equivalent to some value. This is guaranteed by operational membership in some set. From **(forall.elim)** and the fact that operational membership in a set implies operational membership in V^* we can derive the instantiation rule for restricted quantification and the rule for existential introduction.

■(Quantifier laws)

(forall.elim) $(\forall x)\Phi \wedge q \in V^* \Rightarrow \Phi\{x/\varphi\}$

(forall.intro) $\Phi \quad ; \quad (\forall x)\Phi \quad ; \text{ if } x \text{ does not occur free in any implicit assumptions}$

(inst) $(\forall u \in U)\Phi \wedge q \in U \Rightarrow @\{U/q\}$

(exist.intro) $q \in V^* \wedge \Phi\{x/\varphi\} \Rightarrow (\exists x)\Phi$

The fact that operational approximation is a partial order is expressed by the reflexivity and transitivity laws.

■ (Partial order)

(refl) $Q \sqsubseteq Q$

(tran) $Q_0 \sqsubseteq \varphi_1 \wedge \varphi_1 \sqsubseteq Q_2 \Rightarrow Q_0 \sqsubseteq Q_2$

(refl) and **(tran)** b d~~w~~ from the corresponding facts for dtrees.

Substitution of approximate forms for a variable in a third form gives approximate forms and substitution of equivalent forms for a variable in a formula gives an equivalent formula.

■(Substitution)

(subst.term) $\varphi_0 \sqsubseteq \varphi_1 \Rightarrow \varphi\{x/\varphi_0\} \sqsubseteq \varphi\{x/\varphi_1\}$

(subst.opin) $\varphi_0 \cong \varphi_1 \wedge \varphi_0 \in U \Rightarrow \varphi_1 \in U$

(subst.wff) $\varphi_0 \cong \varphi_1 \Rightarrow \Phi\{x/\varphi_0\} \Leftrightarrow \Phi\{x/\varphi_1\}$

(subst.term), and **(subst.opin)** following from the corresponding properties of dtrees. **(subst.wff)** can be derived from **(subst.term)** and **(subst.opin)** using the basic logic of partial terms.

In carrying out proofs, we will use the basic logic of partial terms and operational approximation without explicit mention.

5. 2. 2. In laws

In order to make reasoning in our language completely syntactic we need to provide a language for defining sets of values and rules for reasoning about such sets. As a step in this direction we will specify a collection of basic sets of values and some operations for generating additional sets. We will give some rules for proving operational membership based on the constructions of forms and sets. These rules are analogous to rules for type assignment in a formal **type** system.

The basic sets of values include \emptyset (the empty set), \mathbf{M}_t (the singleton set containing only the empty sequence), \mathbf{D} , \mathbf{O}^D , \mathbf{P} , and \mathbf{C}_o . There may be additional basic subsets of \mathbf{D} depending on the given data structure. For example in the case of S-expressions we will also have \mathbf{D}_{mtl} , \mathbf{D}_{int} , \mathbf{D}_{str} , and \mathbf{D}_{pair} as basic sets. From the basic sets we can obtain additional sets by operations including finite set formation $\{v_1, \dots, v_n\}$, union (\cup), intersection (\cap), formation of finite sequences (*), function space formation ($[\rightarrow]$), concatenation ($[,]$), and selection (1^{st} , r^{st}). Finite set formation, union, and intersection are the usual set-theoretic operations. Formation of finite sequences is described in §3. Concatenation and selection are the lifting of sequence operations to sets of sequences. Thus

$$\begin{aligned} u \in [U_0, U_1] &\Leftrightarrow (\exists u_0 \in U_0, u_1 \in U_1)(u = [u_0, u_1]) \\ u \in 1^{\text{st}}(U) &\Leftrightarrow (\exists u' \in U)(u = 1^{\text{st}}(u')) \\ u \in r^{\text{st}}(U) &\Leftrightarrow (\exists u' \in U)(u = r^{\text{st}}(u')) \end{aligned}$$

Function space formation selects the pfns that compute total functions from the domain component to the range component.

$$P \in [U_0 \rightarrow U_1] \Leftrightarrow (\forall u_0 \in U_0)(\exists u_1 \in U_1)p(u_0) \cong u_1$$

We write $[U_0, \dots, U_n \rightarrow U]$ for $[U_0 \dots \rightarrow [U_n \rightarrow U] \dots]$ and $U_0 \oplus U_1$ for $U_0 \cup U_1$ when $U_0 \cap U_1 = \emptyset$ (disjoint union).

Some examples are

$$\begin{aligned} \emptyset &= \{ \} \\ \mathbf{M}_t &= \{ \}^* = \{ \square \} \\ \mathbf{O} &= \mathbf{O}^D \oplus \mathbf{P} \oplus \mathbf{C}_o \\ \mathbf{V} &= \mathbf{D} \oplus \mathbf{O} \\ \mathbf{V}^+ &= [\mathbf{V}, \mathbf{V}^*] \\ \mathbf{V}^* &= \mathbf{M}_t \oplus \mathbf{V}^+ \end{aligned}$$

The following are some basic laws for operational membership. They will be referred to collectively as “**in laws**”. The **validity of these laws follows** easily from the definitions.

■ (Logical in laws)

$$\varphi \not\in \emptyset$$

$$x \not\in U_0 \wedge U_0 \subset U_1 \Rightarrow x \not\in U_1$$

$$x \not\in (U_0 \cup U_1) \Rightarrow x \not\in U_0 \vee x \not\in U_1$$

$$x \not\in U_0 \wedge x \not\in U_1 \Rightarrow x \not\in (U_0 \cap U_1)$$

■ (In laws for forms)

$$x \not\in V^* \quad ;; x \text{ any variable symbol}$$

$$x \not\in \{x\}$$

$$\lambda x. \varphi \not\in P$$

$$o \not\in [D^* \rightarrow D^*] \quad ;; o \text{ a data operation symbol}$$

$$(\forall x \not\in U_0)(\varphi \not\in U_1) \Rightarrow \lambda x. \varphi \not\in [U_0 \rightarrow U_1]$$

$$f \not\in [U_0 \rightarrow U_1] \wedge x \not\in U_0 \Rightarrow f(x) \not\in U_1$$

$$x \not\in M_t \Rightarrow x \cong mt$$

$$x_0 \not\in U_0 \wedge x_1 \not\in U_1 \Rightarrow [x_0, x_1] \not\in [U_0, U_1]$$

$$x \not\in U \Rightarrow \text{fst}(x) \not\in 1^{\text{st}}(U) \wedge \text{rst}(x) \not\in r^{\text{st}}(U)$$

$$\text{top} \not\in C_o$$

$$c \not\in C_o \Rightarrow c \circ f \not\in C_o$$

$$(\forall c \not\in C_o)(\varphi \not\in U) \Rightarrow \text{note}(c)\varphi \not\in U$$

5.2.3. Some proof schemes

Standard techniques for proof include arguments by cases and induction. Such arguments are valid in *Rum*. Argument by value cases corresponds to computation using **if** and is often used to prove properties involving conditional expressions.

■ (**Vcases**) Since $V^* = M_t \oplus V^+$ to prove that Φ is true for all x we need only prove Φ holds in the case $x \cong mt$ and in the case that $x \not\in V^+$. Formally we have the scheme

$$(\forall x)(x \cong M_t \Rightarrow \Phi) \wedge x \not\in V^+ \Rightarrow \Phi \Rightarrow (\forall x)\Phi$$

A general principle of induction is induction on a well-founded ordering. A special case is rank induction.

■ (**Well-founded induction**) Let U be a sort with well-founded order \prec . For example take U to be **N** and \prec to be $<$ or take U to be the S-expressions and \prec to be the sub-expression relation. We may prove that $\Phi(x)$ holds for $x \in U$ by proving that for any $x \in U$, $\Phi(x)$ follows from the assumption **that** $\Phi(y)$ holds for all y smaller than x . This is formalized by the the following **induction** scheme

$$(\forall x \in U)((\forall y \in U)(y \prec x \Rightarrow \Phi(y)) \Rightarrow \Phi(x)) \Rightarrow (\forall x \in U)(\Phi(x))$$

For example induction on $<$ is the usual course-of-values induction on the natural numbers and induction on the subexpression ordering on S-expressions is the usual S-expression induction.

■ (**Rank induction**) The length **of a sequence** gives rise to a well-founded ordering on a set of sequences and the size of an S-expression gives rise to a well-founded ordering on a set of S-expressions. More generally, let U be a sort and let ρ be a function from U to **N**. ρ is called a **rank function** and for $u \in U$, $p(u)$ is called the rank of u . The ordering \prec_ρ on U defined by

$$y \prec_\rho x \Leftrightarrow \rho(y) < \rho(x)$$

is a well-founded ordering. Induction on rank is formalized by the rank induction scheme

$$(\forall x \in U)((\forall y \in U)(\rho(y) < \rho(x) \Rightarrow \Phi(y)) \Rightarrow \Phi(x)) \Rightarrow (\forall x \in U)(\Phi(x))$$

5.3. Computational laws

The computational laws are based on the computation laws for dtrees. The intuition underlying the computational laws is that computation states and steps can be simulated using forms and operational equivalence. Continuations are represented by forms satisfying $\varphi \in \mathbf{C}_o$ and dtrees are represented by arbitrary forms. In the step rules **meta** variables ranging over continuations (resp. values) are replaced by variables ranging over continuations (resp. values). Begin and return state formation is represented by application. Using the computation laws, computations steps are simulated by (provable) operational equivalence. Thus if φ is closed and returns a value v in the empty context then we can prove $\text{top}(\varphi) \cong \text{top}(\varphi_v)$ where φ_v is the syntactic representation of v . This is made more precise in the appendix.

The computation laws fall into four groups. The first group concerns general laws for sequence operations. These are just the equational laws for a sequence

structure with the empty sequence, concatenation, and projection operations (see §3). The second group concerns embedding the theory of the underlying data structure. The third group concerns equations between forms representing states that correspond to steps building up, capturing, or discarding computation context. The fourth group gives basic laws for application and conditional forms.

- **(Sequence laws)** For a ranging over singleton values (\vee)

$$\text{fst}[a, x] \cong a$$

$$\text{rst}[a, x] \cong x$$

$$\text{fst}(\text{mt}) \cong \text{rst}(\text{mt}) \cong \text{mt}$$

$$x \cong [\text{fst}(x), \text{rst}(x)]$$

$$[\text{mt}, x] \cong x \cong [x, \text{mt}]$$

$$[[x, y], z] \cong [x, [y, z]]$$

The theory of the underlying data structure is embedded by defining a set of forms called *data forms* that are forms syntactically guaranteed to denote sequences from the data domain. We then interpret \cong restricted data forms as $=$ and \in restricted to data forms and subsets of \mathbf{D}^* as \in .

▷ **(Data forms):** Let X be a set of variable symbols. The set of data forms over X is the least set containing X , the data constants, and mt , and such that if $\varphi, \varphi_1, \varphi_2$ are data forms over X and \circ is a data operation then $\circ(\varphi)$, $[\varphi_1, \varphi_2]$, $\text{fst}(\varphi)$, and $\text{rst}(\varphi)$ are data forms over X .

■ **(dform)** For X ranging over data sequences, φ, φ_j data forms over X and A a sort contained in \mathbf{D}^*

• (data) $\circ \in \mathbf{D}^*$

• If $\varphi_0 = \varphi_1$ holds in \mathfrak{D} then $\varphi_0 \cong \varphi_1$.

• (in) If $\varphi \in A$ holds in \mathfrak{D} then $\varphi \in A$.

Notes

The introduction of the notion of data form allows us to view the language of *Rum* as an extension of the usual language associated with such a data structure. Thus we have reduced reasoning about data forms to reasoning in the structure \mathfrak{D} .

We can extend the class of data forms by adding constants defined by pfns that compute functions on the data domain to the list of data operations. This corresponds to enlarging the data structure by adding the defined data operations.

An alternative method of embedding the theory of the underlying data structure is to assume that it is specified by a collection of axioms (and constraints such as initiality) and include this specification in the laws of *Rum*. Some care needs to be taken that the intended interpretation of the specification is not changed in the process.

Endnote.

■(Context sensitive laws) For c, c' ranging over continuations

- (app) $c(\varphi_0(\varphi_1)) \cong (c \circ \text{appi}(\varphi_1))(\varphi_0)$
- (appi) $(c \circ \text{appi}(\varphi_1))(x) \cong (c \circ \text{appc}(x))(\varphi_1)$
- (if) $c(\text{if}(\varphi_0, \varphi_1, \varphi_2)) \cong (c \circ \text{ifi}(\varphi_1, \varphi_2))(\varphi_0)$
- (cart) $c(\text{cart}(\varphi_0, \varphi_1)) \cong (c \circ \text{carti}(\varphi_1))(\varphi_0)$
- (carti) $(c \circ \text{carti}(\varphi_1))(x) \cong (c \circ \text{cartc}(x))(\varphi_1)$
- (fst) $c(\text{fst}(\varphi_0)) \cong (c \circ \text{fstc})(\varphi_0)$
- (rst) $c(\text{rst}(\varphi_0)) \cong (c \circ \text{rstc})(\varphi_0)$
- (note) $c(\text{note}(c)\Omega) \cong c(\Omega)$
- (sw) $c(c'(x)) \cong c'(x)$

■(lam and if reductions)

- (letv) $\text{let}\{x \leftarrow x\}\Omega = (\lambda x. \varphi)(x) \cong \Omega$
- (ifi.nmt) $x \in V^+ \Rightarrow \text{if}(x, \varphi_1, \varphi_2) \cong \varphi_1$
- (ifi.mt) $\text{if}(\text{mt}, \varphi_1, \varphi_2) \cong \varphi_2$

5.3.1. Extensionality and recursion

Our final set of basic laws expresses the essence of operational approximation and equivalence. **(op.ext)** and **(note.abs)** follow from the corresponding dtree properties. Together with the substitution and order laws they say that operational approximation and equivalence are compatible relations (are preserved by form constructions) [Barendregt 1981; p. 50]. **(c.ext)** is just the definition of approximation on dtrees.

■ (Extensionality)

- (op.ext) $f \in \mathbb{O} \wedge g \in \mathbb{O} \wedge (\forall x)(f(x) \sqsubseteq g(x)) \Rightarrow f \sqsubseteq g$
- (note.abs) $(\forall c \in \mathbb{C}_o)(\varphi_0 \sqsubseteq \varphi_1) \Rightarrow \text{note}(c)\varphi_0 \sqsubseteq \text{note}(c)\varphi_1$
- (c.ext) $(\forall c \in \mathbb{C}_o)(c(\varphi_a) \sqsubseteq c(\varphi_b)) \Rightarrow q_a \sqsubseteq q_b \quad ;; c \text{ not free in } \varphi_a, \varphi_b$

The recursion theorem says that the recursion pfn **Rec** computes the least fixedpoint of pfns of the form $\lambda f . \lambda x. \varphi$. It follows from the corresponding theorem for pfns.

■ (Recursion theorem)

- (rec.df) $\mathbf{Rec}(x) \in \mathbf{P}$
- (rec.fix) $F \cong \lambda f. \lambda x. \varphi \text{ A } f \cong \mathbf{Rec}(F) \Rightarrow f(x) \cong \circ$
- (rec.min) $F \cong \lambda f. \lambda x. \varphi \text{ A } f \cong \mathbf{Rec}(F) \text{AF}(g) \sqsubseteq g \Rightarrow f \sqsubseteq g$

Note. (rec.fix) says that pfns **defined** by recursion satisfy the equation obtained by replacing \leftarrow by \cong . The converse of (rec.fix) is not necessarily true since recursion equations contain additional information specifying how computations are to be carried out. For example $f(x) \cong f(x)$ is trivially true but $f(x) \leftarrow f(x)$ implies that f is the everywhere undefined pfn. The recursion theorem extends the least fixed point theorem for the graph model of the lambda calculus [Scott 1976] to a language with control abstractions. **Endnote.**

5.4. Some simple derived laws

To illustrate the use of the basic laws and the logic of partial terms we will derive some simple (and useful) consequences. We begin with two simple exercises.

Exercise. Prove the following corollary to the (**letv**) law.

$$\varphi_0 \in V^* \Rightarrow \text{let}\{x \leftarrow \varphi_0\} \varphi \cong \varphi\{x/\varphi_0\}.$$

Exercise. Prove (rec.df) and (rec.fix) within the first-order theory. Hint use the basic laws for computation, membership, and the extensionality law.

Theorem (nmt): $a \in V \Rightarrow a \not\cong \text{mt}$

Proof (nmt): By the intersection and emptyset inlaws we have

$$a \in V \wedge a \cong \text{mt} \Rightarrow a \in V \cap M_t = \emptyset$$

□ nmt

Value forms give simple syntactic criteria for forms φ_v guaranteed to satisfy $\varphi_v \in V^*$.

D (Vform): Value forms are the least set of forms containing the variable and constant symbols, mt, top, the lambda-forms $(\lambda x. \varphi)$, and containing $[\varphi_{v,1}, \varphi_{v,2}]$, $\text{fst}(\varphi_v)$, $\text{rst}(\varphi_v)$, and $\text{if}(\varphi_v, \varphi_{v,1}, \varphi_{v,2})$. whenever $\varphi_v, \varphi_{v,1}$, and $\varphi_{v,2}$ are value forms.

Theorem (vform): If φ_v is a value form then $\varphi_v \in v^*$.

Proof (vform): by induction on the generation of value forms and the in laws.

□**vform**

We will use **(vform)** without explicit mention to instantiate (unrestricted) universal variables.

Direct application of **(letv)** requires that the argument **expression** be operationally equivalent to a value. It is also valid when the **let** variable occurs in an evaluated position in the body. This will be made precise later. Here we prove a special case in order to illustrate the use of the computation and extensionality laws.

Theorem (idcnv): $\text{let}\{x \leftarrow \varphi\}x \cong \circ$

Proof (idcnv): Note that $\text{let}\{x \leftarrow \varphi\}x = (\lambda x.x)(\varphi)$.

- (i) $c((\lambda x.x)(\varphi)) \cong (c \circ \text{appc}(\lambda x.x))(\varphi) \quad ;; (\text{app,appi})$
- (ii) $(c \circ \text{appc}(\lambda x.x))(x) \cong c((\lambda x.x)(x)) \quad ;; (\text{app,appi})$
- (iii) $c((\lambda x.x)(x)) \cong c(5) \quad ;; (\text{letv})$
- (iv) $c \circ \text{appc}(\lambda x.x) \cong c \quad ;; (\text{op.ext}) \text{ and (iii)}$
- (v) $c((\lambda x.x)(\varphi)) \cong c(\circ) \quad ;; (\text{i}) \text{ and (iv)}$
- (vi) $(\lambda x.x)(\varphi) \cong \circ \quad ;; (\text{v}) \text{ and } (\text{c.ext})$

As advertised, we have used the laws for instantiation and substitution in the above proof without explicit mention. We leave it as an exercise for the reader to fill in the details. □**idcnv**

Another **let-law** is the **let-permutation** rule. This rule allows nested let expressions to be flattened.

Theorem (let.perm):

$$\text{let}\{x \leftarrow \text{let}\{y \leftarrow \varphi_0\}\varphi_1\}\varphi \cong \text{let}\{y \leftarrow \varphi_0\}\text{let}\{x \leftarrow \varphi_1\}\varphi$$

where y is not free in \circ .

Proof (let.perm): Assume y is not free in \circ . Then by the computation laws

- (i) $c(\text{let}\{x \leftarrow \text{let}\{y \leftarrow \varphi_0\}\varphi_1\}\varphi) \cong ((c \circ \text{appc}(\lambda x.\varphi)) \circ \text{appc}(\lambda y.\varphi_1))(\varphi_0)$
- (ii) $c(\text{let}\{y \leftarrow \varphi_0\}\text{let}\{x \leftarrow \varphi_1\}\varphi) \cong (c \circ \text{appc}(\lambda y.\text{let}\{x \leftarrow \varphi_1\}\varphi))(\varphi_0)$
- (iii) $((c \circ \text{appc}(\lambda x.\varphi)) \circ \text{appc}(\lambda y.\varphi_1))(y) \cong (c \circ \text{appc}(\lambda x.\varphi))(\varphi_1)$
 $\cong (c \circ \text{appc}(\lambda y.\text{let}\{x \leftarrow \varphi_1\}\varphi))(y)$

and by **(op.ext)** using y not free \circ

$$(iv) \quad (c \circ \text{appc}(\lambda x.\varphi)) \circ \text{appc}(\lambda y.\varphi_1) \cong c \circ \text{appc}(\lambda y.\text{let}\{x \leftarrow \varphi_1\}\varphi)$$

and by (iv)

$$(v) \quad ((c \circ \text{appc}(\lambda x.\varphi)) \circ \text{appc}(\lambda y.\varphi_1))(\varphi_0) \cong (c \circ \text{appc}(\lambda y.\text{let}\{x \leftarrow \varphi_1\}\varphi))(\varphi_0)$$

Hence by (i),(ii),(v) and **(c.ext)** we are done. $\square_{\text{let.perm}}$

Proofs of the remaining theorems can be found in the appendix. Using **(letv)**, **(op.ext)** and **(rec.min)** (for **(bot)**) we have the following standard properties of partial functions.

Theorem (Laws about functions):

- (lam.abs) $(\forall x \in V^*)(\varphi_0 \cong \varphi_1) \Rightarrow \lambda x.\varphi_0 \cong \lambda x.\varphi_1$
- (op.eta) $f \in \mathbf{O} \Rightarrow \lambda z.f(z) \cong f$
- (cmps.id) $f \in \mathbf{O} \Rightarrow I \circ f \cong f \cong f \circ I$
- (cmps.assoc) $(f \circ g) \circ h \cong f \circ (g \circ h)$
- (bot) $g \in \mathbf{O} \Rightarrow \text{Rec}(\lambda f.\lambda x.f(x)) \sqsubseteq g$

Note. In *Rum* **(op.ext)** is stronger than **(lam.abs)** since the computation domain contains operations other than **pfns** (namely continuations and data operations). There are alternative formulations in which only pfns occur as values and this distinction **goes** away. **(op.eta)** is a limited form of the eta-conversion rule for lambda calculi. This is because in the pure lambda calculus there are only functions so the bounded quantification is trivial. When there are objects other than functions then (eta) only makes sense in the restricted form. **Endnote.**

By **(vcases)** arguments using **(ifmt)** and **(ifnmt)** (and also **(op.ext)** for **(if.lam)**) we have the following properties of **if**.

Theorem (If laws):

- (if.sort) $\varphi_0 \in U_0 \wedge \varphi_1 \in U_1 \Rightarrow \text{if}(z, \varphi_0, \varphi_1) \in U_0 \cup U_1$
- (if.elim) $\text{if}(z, \varphi, \varphi) \cong \varphi$
- (if.perm) $\text{if}(x, \text{if}(y, \varphi_a, \varphi_b), \text{if}(y, \varphi_c, \varphi_d)) \cong \text{if}(y, \text{if}(x, \varphi_a, \varphi_c), \text{if}(x, \varphi_b, \varphi_d))$
- (if.lam) $\lambda x.\text{if}(z, \varphi_1, \varphi_2) \cong \text{if}(z, \lambda x.\varphi_1, \lambda x.\varphi_2)$
- (if.subst) $(\circ \in V^+ \Rightarrow \varphi_1 \cong \varphi_3) \wedge (\circ \cong \text{mt} \Rightarrow \varphi_2 \cong \varphi_4) \wedge \circ \in V^*$
 $\Rightarrow \text{if}(\varphi, \varphi_1, \varphi_2) \cong \text{if}(\varphi, \varphi_3, \varphi_4)$

From the basic computation rules for cart , fst , rst and the in laws we have the following.

Theorem (Cart laws):

$$\begin{aligned} (\text{cart.assoc}) \quad & [[\varphi_0, \varphi_1], \varphi_2] \cong [\varphi_0, [\varphi_1, \varphi_2]] \\ (\text{fst.rst}) \quad & x \in V^+ \Rightarrow \text{fst}[x, y] \cong \text{fst}(x) \wedge \text{rst}[x, y] \cong [\text{rst}(x), y] \end{aligned}$$

Note that the law $x \cong [\text{fst}(x), \text{rst}(x)]$ cannot be generalized to arbitrary terms. A counter example is $\text{note}(c)c$. This is because $c \not\cong c \circ \text{carti}(\text{rst}(x)) \circ \text{fst}_c$.

From the computation rules for noting and switching using $(c.\text{ext})$ we have the following.

Theorem (Note laws):

$$\begin{aligned} (\text{note.triv}) \quad & \text{note}(c)\varphi \cong \varphi \quad ;; c \text{ not free in } \varphi \\ (\text{note.id}) \quad & \text{note}(c)c(\varphi) \cong \varphi \quad ;; c \text{ not free in } \varphi \\ (\text{note.esc}) \quad & c \in C_o \Rightarrow c \circ (\lambda x.\text{note}(c)\varphi) \cong c \circ (\lambda x.\varphi) \\ (\text{note.ren}) \quad & \text{note}(c)\text{note}(c')\varphi \cong \text{note}(c)\varphi\{c'/c\} \end{aligned}$$

■ **Extending laws to extended syntax**) Laws for lambda application and let generalize in a straightforward way to the extended syntax for currying and splitting sequence arguments. For example

$$\begin{aligned} (\text{letv.cart}) \quad & a_1 \in V \mid \vdash \vdash \vdash \mid \exists a \in V \Rightarrow (\lambda [a_1, \dots, a_n]. \text{cart}(a_1, \dots, a_n, y)) \cong \top \\ (\text{letv.curry}) \quad & (\lambda (x_1, \dots, x_n).\varphi)(x_1, \dots, x_n) \cong Q \end{aligned}$$

The scheme for cart associativity says we can treat cart as a multi-ary operation without ambiguity.

5.5. Puzzling with current puzzle.

To illustrate some of the power and also weaknesses of our theory we solve (formally) a puzzle posed in [Queinnec and Séniak 1989]. call-cc is the Scheme control abstraction primitive and can be represented in $\mathcal{R}um$ by

$$D \quad \text{Cwcc} = \lambda f. \text{note}(c)f(c)$$

The puzzle is: what is the meaning of the following two Scheme expressions?

- (i) $(\text{call-cc} \text{ call-cc})$

(ii) $((\text{call-cc } \text{call-cc}) (\text{call-cc } \text{call-cc}))$

Translated into *Rum* we ask the meaning of the corresponding *Rum* expressions.

(i) $\text{Cwcc}(\text{Cwcc})$

(ii) $(\text{Cwcc}(\text{Cwcc}))(\text{Cwcc}(\text{Cwcc}))$

The solution is given by lemmas **(cwcc.2)** and **(cwcc.2.2)** below. **Lemma (cwcc)** is the analog of the `note` law.

Lemma (cwcc): From the `note` law we have (for c ranging over continuations)

$$c(\text{Cwcc}(f)) \cong c(f(c))$$

Lemma (cwcc.2): $\text{Cwcc}(\text{Cwcc}) \cong \text{note}(c)c$

Proof (cwcc.2): We have by two applications of (cwcc) and the sw law

$$\begin{aligned} c(\text{Cwcc}(\text{Cwcc})) &\cong c(\text{Cwcc}(c)) \quad \text{;; by (cwcc)} \\ &\cong c(c(c)) \quad \text{;; by (cwcc)} \\ &\cong c(c) \quad \text{;; by the sw law} \\ &\cong c(\text{note}(c)c) \quad \text{;; by the note law} \end{aligned}$$

Hence by extensionality we are done. $\square_{\text{cwcc.2}}$

Lemma (cwcc.2.2): $(\text{Cwcc}(\text{Cwcc})(\text{Cwcc}(\text{Cwcc})))$ is undefined — describes an infinite computation in every context. Formally we can express this as

$$(\text{Cwcc}(\text{Cwcc})(\text{Cwcc}(\text{Cwcc}))) \not\in V^*.$$

Proof (cwcc.2.2): This will be an informal outline. We take the computation laws seriously as reduction rules and show that in any context the computation sequence for $c(\text{note}(c)c)(\text{note}(c)c)$ is infinite for any c . Let $c_0 = c \circ \text{appi}(\text{note}(c)c)$ and $c_{n+1} = c \circ \text{appc}(c_n)$. Then

(i) $c(\text{note}(c)c)(\text{note}(c)c) \xrightarrow{+} c_0(c_0)$ and

(ii) $c_n(c_n) \xrightarrow{+} c_{n+1}(c_{n+1})$ for any n .

Case (i): By the computation rules we have

$$\begin{aligned} c((\text{note}(c)c)(\text{note}(c)c)) &\xrightarrow{+} (c \circ \text{appi}(\text{note}(c)c))(\text{note}(c)c) \\ &\xrightarrow{+} (c \circ \text{appi}(\text{note}(c)c))(c \circ \text{appi}(\text{note}(c)c)) = c_0(c_0) \end{aligned}$$

□_i

Case (ii): By induction on n and the computation rules we show $c_n(f) \xrightarrow{+} \text{co } (\text{appc}(f))(\text{note}(c)c)$. For $n = 0$

$$c_0(f) = (c \circ \text{appi}(\text{note}(c)c))(f) \xrightarrow{+} (c \circ \text{appc}(f))(\text{note}(c)c)$$

and for $n > 0$

$$\begin{aligned} c_{n+1}(f) &\xrightarrow{+} c(c_n(f)) \xrightarrow{+} c_n(f) \\ &\xrightarrow{+} \text{co appc}(f)(\text{note}(c)c) \quad \text{;; by induction hypothesis} \end{aligned}$$

Now it follows easily that

$$c_n(c_n) \xrightarrow{+} c_{n+1}(\text{note}(c)c) \xrightarrow{+} c_{n+1}(c_{n+1}).$$

□_{ii} 4wcc.2.2

Note. It remains an open question how to axiomatize computation and induction on the length of computation so that arguments like the above can be carried out within the formal theory rather than by appeal to the model. **Endnote.**

5.6. Context motion

The context motion theorem gives some general laws for moving expressions across a special class of contexts we call evaluated *position* contexts. The notion of evaluated position context extends the notion of evaluation context (continuation) by looking inside lets and notes (see also [Felleisen and Friedman 1986, Felleisen 1987]). We will define the notion of evaluated position contexts, state the main theorem and derive some useful consequences to illustrate application of the theorem.

5.6.1. Context motion theorem

□ (**epcx**): The set of evaluated position contexts and the variables trapped by an evaluated position context are defined in Figure 7. Evaluated position context is the least set closed under the constructions in the first column and $\text{trap}(C)$, the variables trapped by the context C is defined in the second column by induction on context construction. $C[\varphi]$ is the result of replacing $[]$ by \circ in the first step of the construction of C . Context equality is modulo alpha-conversion as usual with

C	$\text{trap}(C)$
$[]$	$\{\}$
$C_0(\varphi_1)$	$\text{trap}(C_0)$
$x(C_1)$	$\text{trap}(C_1)$
$[C_0, \varphi_1]$	$\text{trap}(C_0)$
$[x, C_1]$	$\text{trap}(C_1)$
$\text{if}(C_0, \varphi_1, \varphi_2)$	$\text{trap}(C_0)$
$\text{if}(x, C_1, C_2)$	$\text{trap}(C_1) \cup \text{trap}(C_2)$
$\text{fst}(C_0)$	$\text{trap}(C_0)$
$\text{rst}(C_0)$	$\text{trap}(C_0)$
$\text{let}\{z \leftarrow x\}C_0$	$\{z\} \cup \text{trap}(C_0)$
$\text{note}(c)C_0$	$\{c\} \cup \text{trap}(C_0)$

Figure 7. Evaluated position contexts

the restriction that a bound variable with a hole in its scope cannot be renamed. Note that $\text{Frees}([x, \varphi]) \cap \text{trap}(C) = \emptyset$ implies that $C[\varphi] = C[x]\{x/\varphi\}$.

Theorem (context motion): Let C be an evaluated position context and let c range over continuations. Assume $\text{Frees}[\varphi, x, c, k] \cap \text{trap}(C) = \emptyset$ and x, k are not free in C . Then

- (letx) $\text{let}\{x \leftarrow \varphi\}C[x] \cong C[\varphi]$
- (escape) $C[c(\varphi)] \cong c(\varphi)$
- (let.dist) $C[\text{let}\{x \leftarrow \varphi\}\varphi_0] \cong \text{let}\{x \leftarrow \varphi\}C[\varphi_0]$
- (if.dist) $C[\text{if}(\varphi, \varphi_1, \varphi_2)] \cong \text{if}(\varphi, C[\varphi_1], C[\varphi_2])$
- (note.dist) $C[\text{note}(k)\varphi] \cong \text{note}(k)\text{let}\{k \leftarrow k \circ \lambda x.C[x]\}C[\varphi]$

Proof (context motion): The proof of **(letx)** is a straightforward induction on the construction of evaluated position contexts. **(escape)** follows from **(letx)** and the special case of escaping from the argument of a let expression. For **(let.dist)** we observe that $\text{let}\{x \leftarrow \varphi\}\varphi_0 = (\lambda x.\varphi)(\varphi_0)$ and thus we can use **(letx)** followed by **(letv)** to distribute the `let`. **(if.dist, let.dist, note.dist)** are not simple applications of **(letx)** since $\varphi_0, \varphi_1, \varphi_2$ do not need to satisfy the restriction placed on φ .

(if.dist,note.dist) are proved by induction on the number of trapped variables in C. The key is to note that (by induction on construction of C) either C has no trapped variables (is a pure evaluation context) or we can find C_0, C_1 such that $\text{trap}(C_0) = \emptyset$ and $C = C_0[\text{let}\{x \leftarrow v\}C_1]$ or $C = C_0[\text{note}(c)C_1]$. The base case is proved using **(letx)** and the special cases where the context is the argument of a let. The induction cases are proved using the above decomposition of contexts and **(note.rename,note.escape,note.if)** from above. \square context motion

5.6.2. Consequences of the context motion theorem

Corollary (defn.inst): If F is defined by

$$F(x_1, \dots, x_i, \dots, x_n) \leftarrow C[x_i]$$

then

$$F(x_1, \dots, \varphi, \dots, x_n) \cong C[\varphi]$$

Proof (defn.inst):

$$\begin{aligned} F(x_1, \dots, \varphi, \dots, x_n) &\cong \text{let}\{x_i \leftarrow \varphi\} F(x_1, \dots, x_i, \dots, x_n) \quad \text{;; (letx)} \\ &\cong \text{let}\{x_i \leftarrow \varphi\} C[x_i] \quad \text{;; (rec.flx)} \\ &\cong C[\varphi] \quad \text{;; (letx)} \end{aligned}$$

\square defn.inst

Corollary (escape): For c ranging over continuations

$$\begin{aligned} (\text{esc.fun}) \quad (c(\varphi_0))\varphi_1 &\cong c(\varphi_0) \quad \text{;; } C = [\](\varphi_1) \\ (\text{esc.arg}) \quad z(z_1, \dots, z_{j-1}, c(\varphi_j), \dots, \varphi_n) &\cong c(\varphi_j) \\ &\quad \text{;; } C = z(z_1, \dots, z_{j-1}, [\], \dots, \varphi_n) \\ (\text{esc.cart}) \quad [z_1, \dots, z_{j-1}, c(\varphi_j), \dots, \varphi_n] &\cong c(\varphi_j) \\ &\quad \text{;; } C = [z_1, \dots, z_{j-1}, [\], \dots, \varphi_n] \\ (\text{esc.if}) \quad \text{if}(c(\varphi_0), \varphi_1, \varphi_2) &\cong c(\varphi_0) \quad \text{;; } C = \text{if}([\], \varphi_1, \varphi_2) \\ (\text{esc.fst}) \quad \text{fst}(c(\varphi_0)) &\cong c(\varphi_0) \quad \text{;; } c = \text{fst}([]) \\ (\text{esc.rst}) \quad \text{rst}(c(\varphi_0)) &\cong c(\varphi_0) \quad \text{;; } c = \text{rst}([\]) \end{aligned}$$

Corollary (if.dist):

- (if.if) $\text{if}(\text{if}(\varphi_0, \varphi_1, \varphi_2), \varphi_a, \varphi_b) \cong \text{if}(\varphi_0, \text{if}(\varphi_1, \varphi_a, \varphi_b), \text{if}(\varphi_2, \varphi_a, \varphi_b))$
 $\quad \text{;; } C = \text{if}([\], \varphi_a, \varphi_b)$
- (if.or) $\text{if}(\text{or}(\varphi_a, \varphi_b), \varphi_1, \varphi_2) \cong \text{if}(\varphi_a, \varphi_1, \text{if}(\varphi_b, \varphi_1, \varphi_2)) \quad \text{;; (if.if), (if.eval)}$
- (if.fun) $(\text{if}(\varphi_0, \varphi_1, \varphi_2))\varphi \cong \text{if}(\varphi_0, \varphi_1(\varphi), \varphi_2(\varphi)) \quad \text{;; } C = [\](\varphi)$
- (if.arg) $z(z_1 \dots \bullet \underset{\text{zj,if(Qa,Qb,Qc),***Qn}}{z_j}, \text{if}(Qa, Qb, Qc), \dots, Qn)$
 $\cong \text{if}(\varphi_a, z(z_1, \dots, z_j, \varphi_b, \dots, \varphi_n), z(z_1, \dots, z_j, \varphi_c, \dots, \varphi_n))$
 $\quad \text{;; } C = z(z_1, \dots, z_{j-1}, [\], \dots, \varphi_n)$
- (if.cart) $[z_1, \dots, z_j, \text{if}(\varphi_a, \varphi_b, \varphi_c), \dots, \varphi_n]$
 $\cong \text{if}(\varphi_a, [z_1, \dots, z_j, \varphi_b, \dots, \varphi_n], [z_1, \dots, z_j, \varphi_c, \dots, \varphi_n])$
 $\quad \text{;; } C = [z_1, \dots, z_{j-1}, [\], \dots, \varphi_n]$
- (if.let) $Q \in v^* \Rightarrow \text{let}\{x \leftarrow \varphi\}\text{if}(\varphi_0, \varphi_1, \varphi_2) \cong \text{if}(\varphi_0, \text{let}\{x \leftarrow \varphi\}\varphi_1, \text{let}\{x \leftarrow \varphi\}\varphi_2)$
 $\quad \text{;; } C = \text{let}\{x \leftarrow \varphi\}[\], x \text{ not free in } \varphi_0$
- (if.note) $\text{note}(c)\text{if}(\varphi_0, \varphi_1, \varphi_2) \cong \text{if}(\varphi_0, \text{note}(c)\varphi_1, \text{note}(c)\varphi_2)\varphi$
 $\quad \text{;; } C = \text{note}(c)[], c \text{ not free in } \varphi_0$

Corollary (let.dist):

- (let.app) $z(\text{let}\{x \leftarrow \varphi_0\}\varphi_1) \cong \text{let}\{x \leftarrow \varphi_0\}z(\varphi_1) \quad \text{;; } C = z([\])$
- (let.note) $\text{let}\{x \leftarrow \varphi\}\text{note}(c)\varphi_1 \cong \text{note}(c)\text{let}\{x \leftarrow \varphi\}\varphi_1 \quad \text{;; } C = \text{note}(c)[]$
 $\quad \text{;; } c \text{ not free in } \varphi$

Corollary (note.dist):

- (note.fun) $(\text{note}(c)\varphi_0)\varphi_1 \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{appi}(\varphi_1)\}\varphi_0(\varphi_1)$
 $\quad \text{;; } C = [\](\varphi_1)$
- (note.arg) $z(\text{note}(c)\varphi_1) \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{appc}(z)\}z(\varphi_1) \quad \text{;; } C = z([\])$
- (note.test) $\text{if}(\text{note}(c)\varphi_0, \varphi_1, Q2) \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{ifi}(\varphi_1, \varphi_2)\}\text{if}(\varphi_0, \varphi_1, \varphi_2)$
 $\quad \text{;; } C = \text{if}([\], \varphi_1, \varphi_2)$
- (note.carti) $[\text{note}(c)\varphi_0, \varphi_1] \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{carti}(\varphi_1)\}[\varphi_0, \varphi_1]$

;; $C = [[\]], \varphi_1]$

(note.cartc) $[z, \text{note}(c)\varphi_1] \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{cartc}(z)\}[z, \varphi_1]$;; $C = [z, [\]]$

(note.fst) $\text{fst}(\text{note}(c)\varphi_0) \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{fstc}\}\text{fst}(\varphi_0)$;; $c = \text{fst}([\])$

(note.rst) $\text{rst}(\text{note}(c)\varphi_0) \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \text{rstc}\}\text{rst}(\varphi_0)$;; $C = \text{rst}([\])$

Note. The corollaries to **(escape)** and **(note.dist)** parallel the rules for moving context capturing and aborting primitives to the outside of an expression given in [Felleisen and Friedman 1986]. Endnote.

6. Proving properties

In the previous section we showed how to interpret forms as terms in a language for expressing extensional properties of pfns and more generally properties of sequences from the computation domain. The formal semantics of this language was defined and general laws for reasoning in this language were developed. Now we will fix the data structure to be the S-expression structure and use these tools to formulate and prove the extensional properties of pfns expressed informally in §2.

6.1. Vari-ary functions and function schemes

Recall that StrConc is the string concatenation operation defined by

$$\mathbf{D} \quad \text{StrConc}[x, y] \leftarrow \text{StrMk}[\text{StrUn}(x), \text{StrUn}(y)]$$

The fact that StrConc maps pairs of strings to strings is formalized as follows.

Theorem (StrConc.sort): $\text{StrConc} \in [[\mathbf{D}_{\text{str}}, \mathbf{D}_{\text{str}}] \rightarrow \mathbf{D}_{\text{str}}]$

Proof (StrConc.sort): Assume that $a_0 \in \mathbf{D}_{\text{str}}$ and $a_1 \in \mathbf{D}_{\text{str}}$. Then we must show that $\text{StrConc}[a_0, a_1] \in \mathbf{D}_{\text{str}}$. Note that $\text{StrUn}(a_0)$, $\text{StrUn}(a_1)$, and $\text{StrMk}[\text{StrUn}(a_0), \text{StrUn}(a_1)]$ are data forms. Thus we have

$$\begin{aligned} \text{StrConc}[a_0, a_1] &\cong \text{StrMk}[\text{StrUn}(a_0), \text{StrUn}(a_1)] \quad ;; \text{ rec.fix} \\ [\text{StrUn}(a_0), \text{StrUn}(a_1)] &\in \mathbf{D}_{\text{int}}^* \quad ;; \text{ S-expression theory} \\ \text{StrMk}[\text{StrUn}(a_0), \text{StrUn}(a_1)] &\in \mathbf{D}_{\text{str}} \quad ;; \text{ S-expression theory} \end{aligned}$$

$\square \text{StrConc.sort}$

The fact that StrConc is associative and has the empty string ($\text{StrMk}(\text{mt})$) as left and right identity amounts to saying that the domain of strings with StrConc and the empty string constitutes a monoid. This is formalized by **(StrConc.monoid)**.

Theorem (StrConc.monoid): For strings a_0, a_1, a_2

$$\begin{aligned} \text{StrConc}[\text{StrConc}[a_0, a_1], a_2] &\cong \text{StrConc}[a_0, \text{StrConc}[a_1, a_2]] \\ \text{StrConc}[\text{StrM k(mt)}, a_0] &\cong \text{StrConc}[a_0, \text{StrM k(mt)}] \cong a_0 \end{aligned}$$

Proof (StrConc.monoid): Exercise $\square \text{StrConc.monoid}$

Now we turn to the `sit` example. The point here is to show how properties of higher order functions can be thought of as schemes for proving properties of instances.¹ Recall that `sit` is defined by

$$\text{D} \quad \text{Sit}(f, b, x) \leftarrow \text{if}(x, f(\text{fst}(x), \text{Sit}(f, b, \text{rst}(x))), b)$$

The sort of `sit` is given by (`Sit.sort`).

Theorem (`Sit.sort`): For any subdomains A_0 , A_1 of the computation domain

$$\text{Sit} \in [[A_0, A_1 \rightarrow A_1], A_1, A_0^* \rightarrow A_1]$$

Proof (`Sit.sort`): Assume $f \in [A_0, A_1 \rightarrow A_1]$, $b \in A_1$, and $x \in A_0^*$. We will show by induction on x that $\text{sit}(f, b, x) \in A_1$.

Case ($x \cong \text{mt}$): $\text{sit}(f, b, x) \cong b$

Case ($x = [a_0, x_1] \in [A_0, A_0^*]$): by computation $\text{sit}(f, b, x) \cong f(a_0, \text{sit}(f, b, x_1))$. By the induction hypothesis we have $\text{sit}(f, b, x_1) \in A_1$ and by the type of f we have $f(a_0, \text{sit}(f, b, x_1)) \in A_1$. \square `Sit.sort`

Now we use (`Sit.sort`) to deduce the sort of `ListMk`. Recall that `ListMk` is defined by

$$\text{D} \quad \text{ListMk}(x) \vdash \text{Sit}(\lambda x, y. \text{Cons}[x, y], \text{Nil}, x)$$

For any subset A of S-expressions `ListMk` maps sequences from A to lists from A . To formalize this we need to formalize the notion of “lists from A ”.

Definition (`List`): For any set A of S-expressions $\text{List}(A)$ is the least set containing `Nil` and `Cons[a, x]` for each a in A and x in $\text{List}(A)$.

Note that by definition of $\text{List}(A)$ we have

Lemma (`Cons.sort`): $\text{XX}, y. \text{Cons}[x, y] \in [A, \text{List}(A) \rightarrow \text{List}(A)]$. Also, the domain of all lists is just $\text{List}(\mathbb{D}_{\text{sexp}})$.

Now we can formally express the sort of `ListMk` as follows.

Theorem (`ListMk.sort`):

$$A \subset \mathbb{D}_{\text{sexp}} \Rightarrow_{\text{ListMk}} \text{List}(A)$$

¹ A more substantial example is the use of functionals to describe search strategies [Talcott 1985, Appendix B].

Proof (ListMk.sort): by (S_it.sort) and (C_on_s.sort) □ ListMk.sort

Note. An alternative definition of lists from A is $List(A) = \text{ListMk}(A^*)$, extending ListMk to sets of S-expression sequences and we could add ListMk to our list of set constructors. **Endnote.**

Exercise. The inverse of List M k is List U n. Define List U n and show for sequences of S-expressions u and lists x

$$\text{ListUn}(\text{ListMk}(u)) \cong u$$

$$\text{ListMk}(\text{ListUn}(x)) \cong x$$

Exercise. Define ListConc using ListMk in analogy to StrConc and prove that for lists x, y

$$\text{ListConc}(x, y) \cong \text{Append}(x, y)$$

where Append is defined by the usual recursion on lists.

$$\text{D} \quad \text{Append}(x, y) \leftarrow \text{if}(\text{Mtlp}(x), y, \text{Cons}(\text{Car}(x), \text{Append}(\text{Cdr}(x), y)))$$

Exercise. Since \mathbb{D}_{sexp} is finitely and freely generated, and tests for the various sorts are provided, equality between S-expressions SexpEq is computable. Define pfns that test for equality of S-expressions of each sort: IntEq for integers, StrEq for strings, and PairEq for pairs. The test pfns should return the empty sequence if the arguments are not of the correct sort or are not equal. Otherwise they should return one of the arguments. Prove that $\text{IntEq} \in [[\mathbb{D}_{\text{sexp}}, \mathbb{D}_{\text{sexp}}] \rightarrow \mathbb{D}_{\text{sexp}}]$ and that for x, y ranging over S-expressions

$$x \in \mathbb{D}_{\text{int}} \wedge x = y \Rightarrow \text{IntEq}(x, y) \cong x$$

$$\neg(x \in \mathbb{D}_{\text{int}}) \vee x \neq y \Rightarrow \text{IntEq}(x, y) \cong \text{mt}$$

Similarly for StrEq and PairEq. SexpEq is defined by

$$\text{SexpEq}[x, y] \cong \text{or}(\text{IntEq}[x, y], \text{StrEq}[x, y], \text{PairEq}[x, y])$$

SexpEq is not specified for non-S-expression arguments. Prove that for S-expressions x, y

$$\text{SexpEq}[x, y] \cong \begin{cases} x & x = y \\ \text{mt} & x \neq y \end{cases}$$

6.2. Object behaviors.

The set of message-reply sequences constituting an objects observable behavior can be thought of as a tree with edges labeled by messages and nodes labeled by replies. Such a tree is characterized by a function mapping sequences of messages (histories) to a function (the immediate behavior relative to the history) mapping messages (the current message) to replies. Given a behavior function B , we say that a pfn ϑ describes behavior B if there is a family of pfns ϑ_h indexed by sequences of messages h such that $\vartheta_\emptyset = \vartheta$ and $\vartheta_h(m) \cong \vartheta_{[h,m]}, B(h, m)$ for any (acceptable) message sequence h and message m .

Recall that a cell with contents a responds to a get message by returning a and responds to a set message with contents component b by becoming a cell with contents b .

Using the tests **Getmsg** and **Setmsg** to distinguish message types, and the selector **Contents** to extract the contents component of a set message, the cell behavior function $C(a)$ can be specified (by induction on the history) as follows.

Definition (Specification of cell behavior):

$$\text{Getmsg}(m) \Rightarrow C(u)(o)(m) = a \wedge C(a)[m, h](m') = C(a)(h)(m')$$

$$\text{Setmsg}(m) \Rightarrow C(a)(o)(m) = \square \wedge C(a)[m, h](m') = C(\text{Contents}(m))(h)(m')$$

Now we can make the statement that a pfn describes the behavior of a cell with contents a precise by saying it generates the behavior $C(a)$. The correctness of the pfn **Cell** is expressed the following theorem.

Theorem (cell.beh): **Cell(a)** generates the behavior $C(a)$.

Recall that the pfn **Cell(a)** is defined by

$$\begin{aligned} D \quad \text{Cell}(a) &\leftarrow \lambda m. \text{if}(\text{Getmsg}(m), [\text{Cell}(u), a], \\ &\quad \text{if}(\text{Setmsg}(m), \text{Cell}(\text{Contents}(m)), \\ &\quad \text{Cell}(a))) \end{aligned}$$

To get a better understanding of cell behavior we define three auxiliary pfns. Next computes the next state of a cell, Reply computes the reply to a message, and Eff computes the effect of a sequence of messages on the state of a cell.

$$\begin{aligned} D \quad \text{Next}(a, m) &\leftarrow \text{if}(\text{Getmsg}(m), a, \text{if}(\text{Setmsg}(m), \text{Contents}(m), a)) \\ D \quad \text{Reply}(a, m) &\leftarrow \text{if}(\text{Getmsg}(m), a, m) \\ D \quad \text{Eff}(a, h) &\leftarrow \text{if}(h, a, \text{Eff}(\text{Next}(a, \text{fst}(h)), \text{rst}(h))) \end{aligned}$$

Then we have the following useful facts.

Lemma (factoring cell behavior):

- (cell.response) $\text{Cell}(u)(m) \cong [\text{Cell}(\text{Next}(a, m)), \text{Reply}(h, m)]$
- (next.eff) $\text{Eff}(u, [h, m]) = \text{Next}(\text{Eff}(a, h), m)$
- (reply.eff) $\text{Reply}(\text{Eff}(ah), m) = C(u)(h)(m)$

Exercise. Prove the factoring lemma.

Now we can prove (**cell.beh**)

Proof (cell.beh): We must find a family of pfns that satisfy the definition of “generates a behavior”. Take $\vartheta_{a,h} = \text{Cell}(\text{Eff}(a, h))$. Then we have $\vartheta_{a,a} = \text{Cell}(a)$ and

$$\begin{aligned} \vartheta_{a,h}(m) &\cong [\text{Cell}(\text{Next}(\text{Eff}(a, h), m)), \text{Reply}(\text{Eff}(a, h), m)] \\ &\quad ;; \text{ by (cell.response)} \\ &\cong [\text{Cell}(\text{Eff}(a, [h, m])), C(a)(h)(m)] \quad ;; \text{ by (next.eff) and (reply.eff)} \end{aligned}$$

□_{cell.beh}

6.3. The tree product pfns

Now we will prove the equations (**Tpc.Tp**) and (**Tps.Tp**) which express the functional correctness of the tree product pfns Tpc and Tps . The key to these proofs are the equations (**Tc.Tp**) and (**Ts.Tp**), characterizing the functions computed by the auxiliary pfns Tc and Ts . For the reader’s convenience, we repeat the definitions of Tp and friends here.

- ▷ $\text{Tp}(x) \leftarrow \text{if}(\text{Atom}(x), x, \text{Tp}(\text{Car}(x)) * \text{Tp}(\text{Cdr}(x)))$
- ▷ $\text{Tpc}(x) \leftarrow \text{Tc}(x, \text{I})$
- ▷ $\text{Tc}(x, f) \leftarrow \text{if}(\text{Atom}(x), \text{if}(\text{Zerop}(x), 0, f(x)), \text{Tc}(\text{Car}(x), \text{Ta}(x, f)))$
- ▷ $\text{Ta}(x, f) \leftarrow \lambda y. \text{Tc}(\text{Cdr}(x), \text{Td}(y, f))$
- ▷ $\text{Td}(y, f) \leftarrow \lambda z. f(y * z)$
- ▷ $\text{Tps}(x) \leftarrow \text{note}(\text{c})\text{Ts}(x, \text{c})$
- ▷ $\text{Ts}(x, \text{c}) \leftarrow \text{if}(\text{Atom}(x), \text{if}(\text{Zerop}(x), \text{c}(\text{O}), x), \text{Ts}(\text{Car}(x), \text{c}) * \text{Ts}(\text{Cdr}(x), \text{c}))$
- ▷ $\text{Inz}(x) \leftarrow \text{if}(\text{Atom}(x), \text{Zerop}(x), \text{or}(\text{Inz}(\text{Car}(x)), \text{Inz}(\text{Cdr}(x))))$

We begin with a lemma giving the sorts of Inz and Tp and expressing the fact that if a zero occurs in a number tree then the tree product is zero.

Lemma (Tp.Inz): For x ranging over Ntree

$$(\text{Inz.sort}) \quad \text{Inz} \in [\text{Ntree} \rightarrow \mathbf{V}^*]$$

$$(\text{Tp.sort}) \quad \text{Tp} \in [\text{Ntree} \rightarrow \mathbf{N}]$$

$$(\text{Inz.Tp}) \quad \text{Inz}(x) \in \mathbf{V}^+ \Rightarrow \text{Tp}(x) \cong 0$$

Proof (Tp.Inz): an easy induction on number trees. $\square_{\text{Tp.Inz}}$

Functional characterizations of the tree product functions are given by **(tprod)**.

Theorem (tprod): For x ranging over Ntree and c ranging over \mathbb{C}_o

$$(\text{Tp.c.Tp}) \quad \text{Tp.c}(x) \cong \text{Tp}(x)$$

$$(\text{Tc.Tp}) \quad \text{Tc}(x, f) \cong \text{if}(\text{Inz}(x), 0, f(\text{Tp}(x)))$$

$$(\text{Tp.s.Tp}) \quad \text{Tp.s}(x) \cong \text{Tp}(x)$$

$$(\text{Ts.Tp}) \quad \text{Ts}(x, c) \cong \text{if}(\text{Inz}(x), \text{c}(0), \text{Tp}(x))$$

We first prove (Tp.c.Tp) and (Tp.s.Tp) assuming (Tc.Tp) and (Ts.Tp) . The proof of (Tp.s.Tp) was given in §2. We repeat it here to be seen in light of the formal semantics.

Proof (Tp.c.Tp): Assume x ranges over Ntree . Then

$$\begin{aligned} \text{Tp.c}(x) &\cong \text{Tc}(x, \text{I}) \quad \text{;; dfn Tp.c} \\ &\cong \text{if}(\text{Inz}(x), 0, \text{I}(\text{Tp}(x))) \quad \text{;; (Tc.Tp)} \\ &\cong \text{if}(\text{Inz}(x), 0, \text{Tp}(x)) \quad \text{;; (id,letv)} \\ &\cong \text{Tp}(x) \quad \text{;; (Tp.Inz)} \end{aligned}$$

$$\square_{\text{Tp.c.Tp}}$$

Proof (Tp.s.Tp): Assume x ranges over Ntree . Then

$$\begin{aligned} \text{Tp.s}(x) &\cong \text{note}(c)\text{Ts}(x, c) \quad \text{;; dfn Tp.s} \\ &\cong \text{note}(c)\text{if}(\text{Inz}(x), c(0), \text{Tp}(x)) \quad \text{;; (Ts.Tp), (note.abs)} \\ &\cong \text{if}(\text{Inz}(x), \text{note}(c)c(0), \text{note}(c)\text{Tp}(x)) \quad \text{;; (note.if)} \\ &\cong \text{if}(\text{Inz}(x), 0, \text{Tp}(x)) \quad \text{;; (note.triv)} \\ &\cong \text{Tp}(x) \quad \text{;; vcases, (Tp.Inz), (Inz.sort)} \end{aligned}$$

$\square_{\text{Tp}\text{s}.\text{Tp}}$

Now we prove $(\text{Tc}.\text{Tp})$ and $(\text{Ts}.\text{Tp})$. In each case we begin with the righthand side of the equation and obtain the lefthand side by a series of steps that are essentially program transformations (see for example [Scherlis 1981]). Thus, if we had started with $(\text{Tc}.\text{Tp})$ and $(\text{Ts}.\text{Tp})$ as definitions then each step would produce a new definition, preserving the function computed. At the final step we obtain the recursive defining equations. In fact the recursive definitions we originally obtained by just such transformations.

Proof ($\text{Tc}.\text{Tp}$): by Ntree-induction. Assume x ranges over Ntree. Then

$$\begin{aligned}
 & \text{if}(\text{Inz}(x), 0, f(\text{Tp}(x))) \\
 & \cong \text{if}(\text{if}(\text{Atom}(x), \text{Zerop}(x), \text{or}(\text{Inz}(\text{Car}(x)), \text{Inz}(\text{Cdr}(x)))), \quad \text{;; dfn Inz} \\
 & \quad 0, \\
 & \quad f(\text{Tp}(x))) \\
 & \cong \text{if}(\text{Atom}(x), \quad \quad \quad \text{;; (if.if)} \\
 & \quad \text{if}(\text{Zerop}(x), 0, f(\text{Tp}(x))), \\
 & \quad \text{if}(\text{or}(\text{Inz}(\text{Car}(x)), \text{Inz}(\text{Cdr}(x))), 0, f(\text{Tp}(x)))) \\
 & \cong \text{if}(\text{Atom}(x), \quad \quad \quad \text{;; dfn Tp, (if.subst)} \\
 & \quad \text{if}(\text{Zerop}(x), 0, f(x)), \\
 & \quad \text{if}(\text{or}(\text{Inz}(\text{Car}(x)), \quad \text{Inz}(\text{Cdr}(x))), \\
 & \quad \quad 0, \\
 & \quad \quad f(*[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))]))) \\
 & \cong \text{if}(\text{Atom}(x), \quad \quad \quad \text{;; (if.or)} \\
 & \quad \text{if}(\text{Zerop}(x), 0, f(x)), \\
 & \quad \text{if}(\text{Inz}(\text{Car}(x)), \\
 & \quad \quad 0, \\
 & \quad \quad \text{if}(\text{Inz}(\text{Cdr}(x)), \\
 & \quad \quad \quad 0, \\
 & \quad \quad \quad f(*[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))]))))
 \end{aligned}$$

```

 $\cong \text{if}(\text{Atom}(x),$  ;; (letv)
 $\quad \text{if}(\text{Zerop}(x), 0, f(x)),$  ;; (Tp.sort)
 $\quad \text{if}(\text{Inz}(\text{Car}(x)),$ 
 $\quad \quad 0,$ 
 $\quad \quad \text{if}(\text{Inz}(\text{Cdr}(x)),$ 
 $\quad \quad \quad 0,$ 
 $\quad \quad \quad \{\lambda(z)f(*[\text{Tp}(\text{Car}(x)), z])\}\text{Tp}(\text{Cdr}(x))))$ 

 $\cong \text{if}(\text{Atom}(x),$ 
 $\quad \text{if}(\text{Zerop}(x), 0, f(x)),$ 
 $\quad \text{if}(\text{Inz}(\text{Car}(x)),$ 
 $\quad \quad 0,$ 
 $\quad \quad \text{Tc}(\text{Cdr}(x), \lambda(z)f(*[\text{Tp}(\text{Car}(x)), z])))$ 
;; induction hypothesis

 $\cong \text{if}(\text{Atom}(x),$  ;; (letv)
 $\quad \text{if}(\text{Zerop}(x), 0, f(x)),$  ;; (Tp.sort)
 $\quad \text{if}(\text{Inz}(\text{Car}(x)),$ 
 $\quad \quad 0,$ 
 $\quad \quad \{\lambda(y)\text{Tc}(\text{Cdr}(x), \lambda z.f(*[y, z]))\}\text{Tp}(\text{Car}(x)))$ 

 $\cong \text{if}(\text{Atom}(x),$  ;; dfns Ta, Td
 $\quad \text{if}(\text{Zerop}(x), 0, f(x)),$ 
 $\quad \text{if}(\text{Inz}(\text{Car}(s)),$ 
 $\quad \quad 0,$ 
 $\quad \quad \text{Ta}(x, f)(\text{Tp}(\text{Car}(x))))$ 

 $\cong \text{if}(\text{Atom}(x), \text{if}(\text{Zerop}(x), 0, f(x)), \text{Tc}(\text{Car}(x), \text{Ta}(x, f)))$ 
;; induction hypothesis

 $\cong \text{Tc}(x, f)$  ;; dfn Tc

```

$\square_{\text{Tc}, \text{Tp}}$

Proof (Ts.Tp): by Ntree-induction. Assume x ranges over $Ntree$ and c ranges over \mathbb{C}_o . Then

$\text{if}(\text{Inz}(x), c(0), \text{Tp}(x))$

$\cong \text{if}(\text{Atom}(x),$;; as in proof of ($\text{Tp}(\text{Car}(x))$, $\text{Tp}(\text{Cdr}(x))$)
 $\text{if}(\text{Zerop}(x), c(0), x)$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $c(0),$
 $\text{if}(\text{Inz}(\text{Cdr}(x)),$
 $c(0),$
 $*[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))]))$

$\cong \text{if}(\text{Atom}(x),$;; (esc)
 $\text{if}(\text{Zerop}(x), c(0), x),$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $c(0),$
 $\text{if}(\text{Inz}(\text{Cdr}(x)),$
 $*[\text{Tp}(\text{Car}(x)), c(0)],$
 $*[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))]))$

$\cong \text{if}(\text{Atom}(x),$;; (if.dist)
 $\text{if}(\text{Zerop}(x), c(0), x)$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $c(0),$
 $*[\text{Tp}(\text{Car}(x)), \text{if}(\text{Inz}(\text{Cdr}(x)), c(0), \text{Tp}(\text{Cdr}(x)))]])$

$\cong \text{if}(\text{Atom}(x),$;; induction hypothesis
 $\text{if}(\text{Zerop}(x), c(0), x),$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $c(0),$
 $*[\text{Tp}(\text{Car}(x)), \text{Ts}(\text{Cdr}(x), c)])$

$\cong \text{if}(\text{Atom}(x),$;; (esc), (if.dist)
 $\text{if}(\text{Zerop}(x), c(0), x),$
 $*[\text{if}(\text{Inz}(\text{Car}(x)), c(0), \text{Tp}(\text{Car}(x))), \text{Ts}(\text{Cdr}(x), c)])$
 ;; as above

$\cong \text{if}(\text{Atom}(x),$;; induction hypothesis
 $\text{if}(\text{Zerop}(x), c(0), x),$
 $*[\text{Ts}(\text{Car}(x), c), \text{Ts}(\text{Cdr}(x), c)])$

$\cong \text{Ts}(x, c)$;; dfn Ts

Ts.Tp

6.4. Streams and coroutines.

Streams and coroutines are mechanisms for generating elements of a sequence as they are needed. They can be characterized extensionally by the sequences they generate. For simplicity we will concentrate on infinite sequences (thus avoiding the need to consider the case of the empty stream).

6.4.1. Streams

Recall that in *Rum*, streams are represented by pfns which when queried (applied to the empty sequence) return a pfn representing the rest of the stream together with the next element of the stream. We begin by formalizing the notion of sequence generated by a stream.

Definition (Streams generating sequences.): Let σ be an infinite sequence of elements from the computation domain ($a \in [\mathbb{N} \rightarrow V]$). A pfn s is a stream generating σ if there is a sequence of pfns $\vartheta(j)$ such that $s \cong \vartheta(0)$ and $\vartheta(j)(\text{mt}) \cong [\vartheta(j+1), \sigma(j)]$ for all $j \in \mathbb{N}$. We say that ϑ is the sequence of tails of s , $\vartheta(j)$ is the j -th tail of s , and $\sigma(j)$ is the j -th element of s .

Note that if s is a stream generating σ with sequence of tails ϑ then $\vartheta(j)$ is a stream generating $\lambda n. \sigma(n+j)$ with sequence of tails $\lambda n. \vartheta(n+j)$ for each $j \in \mathbb{N}$.

We will use this characterization of streams to formulate and prove the correctness of the pfn **Sieve** discussed in §2. Recall that **Sieve** is defined by

- D $\text{Sieve}(\text{mt}) \leftarrow \text{Sift}(\text{Ints}(2))(\text{mt})$
- D $\text{Ints}(n)(\text{mt}) \leftarrow [\text{Ints}(n+1), n]$
- D $\text{Filter}(p, in)(\text{mt}) \leftarrow \text{let } \{ [n, in] \leftarrow in(\text{mt}) \}$
 $\quad \text{if}(\text{Divp}(p, n), \text{Filter}(p, in)(\text{mt}), [\text{Filter}(p, in), n])$
- D $\text{Sift}(in)(\text{mt}) \leftarrow \text{let } \{ [in, p] \leftarrow in(\text{mt}) \} [\text{Sift}(\text{Filter}(p, in)), p]$

where for numbers p, n the expression $\text{Divp}(p, n)$ is true iff p divides n . Let Primes be the sequence of prime numbers listed in increasing order. The correctness of **Sieve** is expressed as follows.

■ (**Correctness of Sieve.**) **Sieve** generates Primes .

Proof (Correctness of Sieve.): Since **Sieve** $\cong \text{Sift}(\text{Ints}(2))$, to prove the correctness of **Sieve** we need only define sequences giving the elements and tails of $\text{Sift}(\text{Ints}(2))$ and verify that the elements sequence is indeed Primes .

We begin by defining some number-theoretic functions. These definitions can easily be transformed into pfn definitions. We leave them in the world of mathematics to separate the work which is purely number theory from that which

deals with pfns. In the following i, j, k, m, n will range over \mathbb{N} , σ will range over sequences of numbers – $\sigma \in [\mathbb{N} \rightarrow \mathbb{N}]$, and $m \mid n$ is the predicate “ m divides n ”. Define the functions Ix , $Index$, and $\sigma^{(\neg m)}$ for each $m \in \mathbb{N}$ by

$$\begin{aligned} Ix(\sigma, m, j) &= \mu\{k \geq j\} \neg(m \mid \sigma(k)) \\ Index(\sigma, m, 0) &= Ix(\sigma, m, 0) \\ Index(\sigma, m, n+1) &= Ix(\sigma, m, Index(\sigma, m, n) + 1) \\ \sigma^{(\neg m)}(k) &= \sigma(Index(\sigma, m, k)) \end{aligned}$$

Thus $Ix(\sigma, m, j)$ is the least number k such that $k \geq j$ and m does not divide $\sigma(k)$. $Index(\sigma, m, j)$ is the number k such that m does not divide $\sigma(k)$ and such that there are exactly j elements $a(i)$ with $i < k$ and m not dividing $\sigma(i)$. $\sigma^{(\neg m)}$ is the subsequence of elements of σ not divisible by m . We will only apply these functions in situations where σ is a sequence with infinitely many elements not divisible by m so we need not specify their behavior elsewhere. Define the sequences $Pint(n)$ and Pr by

$$\begin{aligned} Pint(0) &= \lambda(i)i + 2 \\ Pint(n+1) &= Pint(n)^{(\neg Pr(n))} \\ Pr(j) &= Pint(j, 0) \end{aligned}$$

Then $Pint(n)$ is the sequence of numbers not divisible by $Pr(j)$ for all $j < n$, $Pr(0) = 2$, and $Pr(j)$ is the j -th prime. Note also that there are infinitely many elements of $Pint(n)$ not divisible by $Pr(n)$.² Thus to prove the correctness of **Sieve** we need only find a sequence of pfns $\vartheta(j)$ such that $\vartheta(0) \cong \text{Sift}(\text{Ints}(2))$ and $\vartheta(n)(\text{mt}) \cong [\vartheta(n+1), Pr(n)]$. Define the sequence of pfns $P(n)$ by

D $P(n) \leftarrow \text{if}(\text{Zerop}(n), \text{Ints}(2), \text{let}\{[s, m] \leftarrow P(n-1)\} \text{Filter}(m, P(n-1)))$

Lemma (P.Pint): For all n , $P(n)$ is a stream generating $Pint(n)$.

The correctness of **Sieve** now follows from the lemma (**Sift.sifts**).

Lemma (Sift .sifts): $\text{Sift}(\text{Ints}(2))$ is a stream generating Pr with j -th tail $\text{Sift}(P(j))$.

Proof (Sift.sifts): We must show $\text{Sift}(P(j))(\text{mt}) \cong [\text{Sift}(P(j+1)), Pr(j)]$. By (P.Pint) there is a stream s such that $P(j)(\text{mt}) \cong [s, Pr(j)]$ and by the definition

² These two sentences are where the main number-theoretic work is hidden.

of **Filter** we have $\text{Filter}(Pr(j), P(j)) \cong \text{Filter}(Pr(j), s)$. Hence by the definition of **Sift** and P we have

$$\text{Sift}(P(j))(mt) \cong [\text{Sift}(\text{Filter}(Pr(j), P(j))), Pr(j)] \cong [\text{Sift}(P(j+1)), Pr(j)]$$

□ Sift.sifts

All that remains is to prove (**P.Pint**). We first prove some lemmas about **Ints** and **Filter**.

Lemma (Ints.num): $\text{Ints}(i)$ generates the sequence $\lambda(n)n + i$, the sequence of numbers greater than or equal to i .

Proof (Ints.num): The j -th tail of $\text{Ints}(i)$ is $\text{Ints}(i+j)$. □ Ints.num

Lemma (Filter .fil): Let σ be a sequence with infinitely many elements not divisible by m and let $in(i)$ be the sequence of tails of a stream generating σ .

$$\text{Filter}(min(i))(mt) \cong [\text{Filter}(min(Ix(\sigma, m, i) + 1)), \sigma(Ix(\sigma, m, i))]$$

Proof (Filter .fil): by induction on $k = Ix(\sigma, m, i) - i$

Case ($k = 0$): $\text{Divp}(m, a(i))$ is false and by definition of **Filter**

$$\text{Filter}(m, in(i))(mt) \cong [\text{Filter}(m, in(i+1)), o(i)]$$

Case ($k > 0$): : Then $\text{Divp}(m, \sigma(i))$ is true and by definition of **Filter** and induction hypothesis

$$\begin{aligned} \text{Filter}(m, in(i))(mt) &\cong \text{Filter}(m, in(i+1))(mt) \\ &\cong [\text{Filter}(m, in(Ix(\sigma, m, i) + 1)), \sigma(Ix(\sigma, m, i))] \end{aligned}$$

□ Filter.fil

Lemma (Filter.filters): Let σ be a sequence with infinitely many elements not divisible by m and let s be a stream generating σ with sequence of tails in . Then $\text{Filter}(m, s)$ is a stream generating $\sigma^{(\neg m)}$ with sequence of tails out where

$$out(0) \cong \text{Filter}(m, s)$$

$$out(j+1) \cong \text{Filter}(m, in(Index(\sigma, m, j) + 1)))$$

Proof (Filter.filters): We must show for each j

$$out(j) \cong [out(j+1), \sigma^{(\neg m)}(j)].$$

This follows easily from **(Filter .fil)** and the properties of *Index*, and $\sigma^{(\neg m)}$ given above.

Case ($j = 0$): Let $k = Ix(\sigma, m, 0)$ then

$$\begin{aligned} Index(a, m, 0) &= k \quad \text{;; defn } Index \\ \sigma^{(\neg m)}(0) &= a(k) \quad \text{;; defn } \sigma^{(\neg m)} \\ out(0)(\text{mt}) &\cong \text{Filter}(m, in(0))(\text{mt}) \cong [\text{Filter}(m, in(k+1)), \sigma(k)] \quad \text{;; (Filter Al)} \end{aligned}$$

Case ($j + 1$): Let $k = Ix(\sigma, m, Index(\sigma, m, j) + 1)$ then

$$\begin{aligned} Index(\sigma, m, j + 1) &= k \quad \text{;; defn } Index \\ \sigma^{(\neg m)}(j+1) &= a(k) \quad \text{;; defn } \sigma^{(\neg m)} \\ out(j+1)(\text{mt}) &\cong \text{Filter}(m, in(Index(\sigma, m, j) + 1))(\text{mt}) \\ &\cong [\text{Filter}(m, in(k + 1)), a(k)] \quad \text{;; (Filter.fil)} \end{aligned}$$

□ Filter Alters

Now we are ready to prove the *Pint* lemma.

Proof (P.Pint): by induction on n . The **case $n = 0$** is by **(Ints.ints)**. For $n > 0$, assume $P(n - 1)$ is a stream generating $Pint(n - 1)$. By **(Filter.filters)** and the definitions of *Pint*, *Pr* we have $P(n) \cong \text{Filter}(Pr(n-1), P(n-1))$ is a stream generating *Pint(n)*. □ **P.Pint**

This completes the proof of correctness of **Sieve**. □ **Correctness of Sieve**.

6.4.2. Coroutines

The next element of the sequence generated by a coroutine is obtained by resuming the coroutine. The coroutine computes the next element and resumes its resumer with the value of the next element in a context that represents the remainder of the computation.

To formalize the notion of coroutine and of sequence generated by a coroutine we begin with the resumption pfn **Resume**. **Resume** notes the current context, discards it (by applying **top**), and calls the resumee with the noted context and possibly additional information as argument.

$$\text{D} \quad \text{Resume}[p, v] \leftarrow \text{note}(c)\text{top}(p[c, v])$$

Resumption is used with no additional arguments to get the next element from a coroutine generating a sequence. It is also used by the generating coroutine to

return the element **to** its resumer. Thus it is common for resumption of a generator coroutine g by a consumer c , to reduce to resumption of c with the next element x in a context g' representing the remainder of the generator computation. In this case resumption of g is equivalent to $[\text{top} \circ g', x]$. This property of co-operating pairs of resumptions is expressed by the following theorem.

Theorem (res.res):

$$\text{Resume}(\lambda x.p(\text{Resume}[x, z])) \cong [\text{top} \circ p, z]$$

Proof (res.res):

$$\begin{aligned} \text{Resume}(\lambda x.p(\text{Resume}[x, z])) &\cong \\ &\cong \text{note}(c_0) \text{top}((\lambda x.p(\text{Resume}[x, z]))(c_0)) \quad \text{;; dfn Resume} \\ &\cong \text{note}(c_0) \text{top}(p(\text{Resume}[c_0, z])) \quad \text{;; (letv)} \\ &\cong \text{note}(c_0) \text{top}(p(\text{note}(c_1)c_0[c_1, z])) \quad \text{;; dfn Resume} \\ &\cong \text{note}(c_0) \text{note}(c_1) \text{top}(p(c_0[c_1 \circ (\text{top} \circ p), z])) \quad \text{;; (note.dist)} \\ &\cong \text{note}(c_0) \text{note}(c_1) c_0[\text{top} \circ p, z] \quad \text{;; (esc.c) twice} \\ &\cong \text{note}(c_0) c_0[\text{top} \circ p, z] \quad \text{;; (note.triv)} \\ &\cong [\text{top} \circ p, z] \quad \text{;; (note.id)} \end{aligned}$$

Now we can formalize the notion of coroutine generating a sequence.

Definition (Coroutines generating sequences): Let σ be an infinite sequence of elements from the computation domain. co is a coroutine generating σ if there is a sequence of pfns $\vartheta(j)$ such that $\vartheta(0) \cong co$ and for all $j \in \mathbb{N}$

$$\text{Resume}(\vartheta(j)) \cong [\text{top} \circ \vartheta(j + 1), \sigma(j)].$$

We say that ϑ is the sequence of tails of co , $29(j)$ is the j -th tail of co , and $a(j)$ is the j -th element of co .

We will now use the resumption theorem and this characterization of coroutines to formulate and prove the correctness of the string-transforming pfn C32. Recall from §2 that C32 and its auxiliaries C32a, C32b, C32c, C32d satisfy

■(C32.aux)

$$\text{C32}(\text{in})[\text{out}] \cong \text{C32a}(\text{out})(\text{Resume}(\text{in}))$$

$$\text{C32a}(\text{out})[\text{in}, w] \cong \text{let}\{[x_0, x_1, x_2] \leftarrow \text{StrUn}(w)\}$$

$$\quad \quad \quad \text{C32b}(\text{in}, x_2)(\text{Resume}[\text{out}, \text{StrMk}[x_0, x_1]])$$

$$\text{C32b}(\text{in}, x_2)[\text{out}] \cong \text{C32c}(\text{out}, x_2)(\text{Resume}(\text{in}))$$

$$\text{C32c}(\text{out}, x_2)[\text{in}, w] \cong \text{let}\{[x_3, x_4, x_5] \leftarrow \text{StrUn}(w)\}$$

$$\quad \quad \quad \text{C32d}(\text{in}, x_4, x_5)(\text{Resume}[\text{out}, \text{StrMk}[x_2, x_3]])$$

$$\text{C32d}(\text{in}, x_4, x_5)[\text{out}] \cong \text{C32}(\text{in})(\text{Resume}[\text{out}, \text{StrMk}[x_4, x_5]])$$

Theorem (C32.correctness): Let χ be a sequence of characters, let $\chi^{(3)}$ be the sequence of strings of length three such that

$$\chi^{(3)}(i) = \text{StrMk}[\chi(3i), \chi(3i + 1), \chi(3i + 2)],$$

and let $\chi^{(2)}$ be the sequence of strings of length two such that

$$\chi^{(2)}(i) = \text{StrMk}[\chi(2i), \chi(2i + 1)]$$

Then for any coroutine co generating $\chi^{(3)}$, $\text{C32}(co)$ is a coroutine generating $\chi^{(2)}$.

Proof (C32.correctness): Let $\chi, \chi^{(3)}, \chi^{(2)}$ be as in the statement of the theorem and let co be a coroutine generating $\chi^{(3)}$ with tails $\text{in}(j)$ for $j \in \mathbb{N}$. We want to show that there is a sequence of pfns $\text{out}(j)$ such that $\text{C32}(co) \cong \text{out}(0)$ and $\text{Resume}(\text{out}(j)) \cong [\text{top} \circ \text{out}(j+1), \chi^{(2)}(j)]$ for $j \in \mathbb{N}$. Define $\text{out}(j)$ as follows:

$$\text{out}(3j) \cong \text{C32}(\text{in}(2j))$$

$$\text{out}(3j + 1) \cong \text{C32b}(\text{in}(2j + 1), \chi(6j + 2))$$

$$\text{out}(3j + 2) \cong \text{C32d}(\text{in}(2j + 2), \chi(6j + 4), \chi(6j + 5))$$

Thus we need only prove the following three equations

$$(\text{out.i}) \quad \text{Resume}(\text{C32}(\text{in}(2j))) \cong [\text{top} \circ \text{C32b}(\text{in}(2j + 1), \chi(6j + 2)), \chi^{(2)}(3j)]$$

$$(\text{out.ii}) \quad \text{Resume}(\text{C32b}(\text{in}(2j + 1), \chi(6j + 2)))$$

$$\cong [\text{top} \circ \text{C32d}(\text{in}(2j + 2), \chi(6j + 4), \chi(6j + 5)), \chi^{(2)}(3j + 1)]$$

$$(\text{out.iii}) \quad \text{Resume}(\text{C32d}(\text{in}(2j + 2), \chi(6j + 4), \chi(6j + 5)))$$

$$\cong [\text{top} \circ \text{C32}(\text{in}(2j + 2)), \chi^{(2)}(3j + 2)]$$

We will use the following simple facts which follow easily from the definition of **Resume** and (**C32.aux**).

- (res.top) $\text{Resume}(p) \cong \text{Resu me(top o } p)$
- (in.top) $\text{C32}(in) \cong \text{C32}(\text{top o in})$
- (in.top.b) $\text{C32b}(in, x) \cong \text{C32b}(\text{top o in, x})$
- (in.top.d) $\text{C32d}(in, x, y) \cong \text{C32d}(\text{top o in, x, y})$

Let $\chi, \chi^{(i)}$ be as in the statement of the theorem and assume $in(j) \ i \in S$ a sequence of tails of a co-routing generating $\chi^{(3)}$. Thus we have

$$(in.hyp) \quad \text{Resume}(in(j)) \cong [\text{top o in}(j + 1), \chi^3(j)]$$

Proof (out.i):

$$\begin{aligned} \text{Resume}(\text{C32}(in(2j))) &\cong \\ &\cong \text{Resume}(\lambda out. \text{C32a}(out)(\text{Resume}(in(2j)))) \quad ;; (\text{C32.aux}) \\ &\cong \text{Resume}(\lambda out. \text{C32a}(out)[\text{top o in}(2j + 1), \chi^{(3)}(2j)]) \quad ;; (\text{hyp.in}) \\ &\cong \text{Resume}(\lambda out. \text{C32b}(in(2j + 1), \chi(6j + 2))(\text{Resume}[out, \chi^{(2)}(3j)])) \\ &\quad ;; (\text{C32.aux}), (\text{in.top.b}), \text{ defn } \chi^{(i)} \\ &\cong [\text{top o C32b}(in(2j + 1), \chi(6j + 2)), \chi^{(2)}(3j)] \quad ;; (\text{res.res}) \end{aligned}$$

□ out.i

Proof (out.ii):

$$\begin{aligned} \text{Resume}(\text{C32b}(in(2j + 1), \chi(6j + 2))) &\cong \\ &\cong \text{Resume}(\lambda out. \text{C32c}(out, \chi(6j + 2))(\text{Resume}(in(2j + 1)))) \quad ;; (\text{C32.aux}) \\ &\cong \text{Resume}(\lambda out. \text{C32c}(out, \chi(6j + 2))[\text{top o in}(2j + 2), \chi^{(3)}(2j + 1)]) \\ &\quad ;; (\text{hyp.in}) \\ &\cong \text{Resume}(\lambda out. \text{C32d}(in(2j + 2), \chi(6j + 4), x_5)(\text{Resume}[out, \chi^{(2)}(3j + 1)])) \\ &\quad ;; (\text{C32.aux}), (\text{in.top.d}), \text{ defn } \chi^{(i)} \\ &\cong [\text{top o C32d}(in(2j + 2), \chi(6j + 4), \chi(6j + 5)), \chi^{(2)}(3j + 1)] \quad ;; (\text{res.res}) \end{aligned}$$

□ out.ii

Proof (out.iii):

$$\begin{aligned}
 & \text{Resume}(\text{C32d}(in(2j + 2), \chi(6j + 4), \chi(6j + 5))) \\
 & \cong \text{Resume}(\lambda out. \text{C32}(in(2j + 2))(\text{Resume}[out, \chi^{(2)}(3j + 2)])) \\
 & \quad \text{;; (C32 .aux), defn } \chi^{(i)} \\
 & \cong [\text{topoC32}(in(2j + 2)), \chi^{(2)}(3j + 2)] \quad \text{;; (res.res)}
 \end{aligned}$$

□**out.iii** □ **C32.correctness**

6.4.3. Remarks

We have formalized useful but particularly simple notions of stream and coroutine. It is easy to see that pfns can represent much more complex stream-like objects and more intricate patterns of coroutine interaction. We refer the reader to the Common Lisp manual [Steele 1984] for more elaborate notions of stream and to [Kahn and McQueen 1977] for a more general notion of systems of coroutines.

7. Derived properties and programs

A set of derived properties is a uniformly computable set of properties of computations described by programs. By uniformly computable we mean computed by a scheme for recursion on the structure of computations. For example *simple* derived properties are computed by composing **a** sequence of parameter functions corresponding to the rules used in generating **a computation** sequence and applying this composition to an initial **value**. **Many more elaborate** schemes are possible. A derivation map converts **programs into derived programs** that compute **derived properties** of the original program. The uniformity requirement for derived properties means that derivation maps are computable operations on programs determined by the recursion schemes. The value of derivation maps is that they convert **intensional properties of programs into** extensional properties of derived programs. **This allows us to apply methods for proving** properties of the function computed by a program to proving properties of the computations described. The term “derived program” is due to McCarthy (private communication) and derived programs are related to work of [Wegbreit 1975]. The idea of derived property is a special case of non-standard interpretation in which we reinterpret program primitives to compute some property of the computation described the standard interpretation. Some forms of abstract interpretation (cf. [Cousot et Cousot 77], [Jones and Mycroft 86], [Abramsky and Hankin 87]) can also be expressed as derived properties.

To illustrate the basic ideas we will focus on simple derived properties, henceforth known as s.d.p.s. We begin with some examples of simple derived properties. Then we formally define the set of s.d.p.s and give some properties of this set. Finally we define a derivation map for s.d.p.s and use it to analyze intensional properties of the tree product pfns. To simplify matters we will assume that we are working over a data structure that contains the S-expression structure.

7.1. Examples of simple derived properties

Two examples of s.d.p.s are $\text{count}(O)$ for a set of rules O and trace . $\text{count}(O)$ is the number of times one of the rules in O is applied in generating a computation sequence and trace is the list of (codes for) the rules applied. Returning to the example computation sequence Σ_{ex} (which we repeat here in Figure 8 for the reader’s convenience) we have

$$\text{count}(\{\text{Atom}\}, \Sigma_{\text{ex}}) = 0, \quad \text{count}(\{\text{Zerop}\}, \Sigma_{\text{ex}}) = 1, \quad \text{count}(\{\text{sw}\}, \Sigma_{\text{ex}}) = 1,$$

and

$$\text{trace}(\Sigma_{\text{ex}}) = \langle \widehat{\text{if}}, \widehat{\text{app}}, \widehat{\text{sym}}, \widehat{\text{appi}}, \widehat{\text{sym}}, \widehat{\text{Zerop}}, \widehat{\text{ifi}}, \widehat{\text{app}}, \widehat{\text{sym}}, \widehat{\text{appi}}, \widehat{\text{sym}}, \widehat{\text{sw}} \rangle.$$

where ξ maps x to 0 and c to top

Figure 8. Example computation sequence

where $\widehat{\text{if}}$, $\widehat{\text{app}}$, etc. denote S-expressions coding the symbols if, app, etc.

7.2. **Definition of simple derived property**

An s.d.p. is defined by giving an initial value and a set of unary functions (primitive derivations), one for each computation rule. Given a set of primitive derivations we define the derived property function for a computation sequence to be the composition of the primitive derivations corresponding to the rules used in generating the sequence. The derived property of a given computation sequence is then computed by applying its derived property function to the initial value. The use of derived property functions allows segments of a computation sequence to be treated independently. For a segment contained in a larger sequence the initial value is interpreted as the derived property for the computation up to the beginning of the segment. Applying the derived property function for the segment to this value produces the derived property for the computation continued to the end of the segment.

D (Simple derived properties.): Let Rules be the set of (S-expression codes for) the computation rules

$$\text{Rules} = \{\widehat{\text{sym}}, \widehat{\text{mt}}, \widehat{\text{top}}, \widehat{\text{lam}}, \widehat{\text{note}}, \widehat{\text{if}}, \widehat{\text{ifi}}, \widehat{\text{app}}, \widehat{\text{appi}}, \\ \widehat{\text{cart}}, \widehat{\text{carti}}, \widehat{\text{cartc}}, \widehat{\text{fst}}, \widehat{\text{fstc}}, \widehat{\text{rst}}, \widehat{\text{rstc}}, \widehat{\text{pfn}}, \widehat{\text{sw}}, \widehat{o} \mid o \in \Theta^D\}.$$

Let A be a subset of V and let \mathbf{D} be a pfn mapping Rules to pfns computing unary functions on A .

$$\mathbf{D} \in [Rules \rightarrow A \rightarrow A]$$

We call \mathbf{D} a primitive derivation over A and Σ^D is the primitive derivation corresponding to \widehat{r} for each $\widehat{r} \in \text{Rules}$. If Σ is the computation sequence $[\zeta_0 \xrightarrow{r_1} \dots \xrightarrow{r_k} \zeta_k]$, then Σ^D , the derived function of Σ determined by \mathbf{D} , is the composition $D(\widehat{r}_k) \circ \dots \circ D(\widehat{r}_1)$. The s.d.p. of Σ determined by $a \in A$ and \mathbf{D} is $\Sigma^D(a)$.

Lemma (count.sdp): count is an s.d.p.

Proof (count.sdp): Let O be a set of rules and let \mathbf{D} be the primitive derivation over the integers defined by

$$D(\widehat{r}) = \begin{cases} \text{Add1} & \text{if } r \in O \\ I & \text{if } r \notin O. \end{cases}$$

Then $\text{count}(O, \Sigma) = \Sigma^D(0)$ and thus $\text{count}(O)$ is the s.d.p. determined by \mathbf{D} and O . $\square_{\text{count.sdp}}$

Lemma (trace.sdp): trace is an s.d.p.s.

Proof (trace.sdp): Let \mathbf{D} be the primitive derivation over lists of integers defined by

$$\mathbf{D} = \lambda r. \lambda d. \text{Append}(d, \text{ListMk}(r))$$

Then $\text{trace}(\mathcal{C}) = \Sigma^D(\text{Nil})$ and thus trace is the s.d.p. determined by \mathbf{D} and Nil . $\square_{\text{trace.sdp}}$

Theorem (trace.universal): The trace of a computation sequence contains all the information about the computation sequence available for computing s.d.p.s. In particular, there is a pfn Derive such that for any $A \subset V$, any $a \in A$, and any primitive derivation \mathbf{D} over A

$$\text{Derive}(\mathbf{D}a, \text{trace}(\mathcal{C})) = \Sigma^D(a)$$

Proof (trace.universal): Take $\text{Derive}(\mathbf{D}, a, x) \cong \text{Sit}(\mathbf{D}, a, \text{ListUn}(x))$ and show by induction on length of Σ that the specification for Derive is satisfied.

$\square_{\text{trace.universal}}$

When computations are defined by recursion and carried out on a stack-based machine, an important property is the maximum stack depth required in a computation. In a continuation-based model, the analog to stack depth is continuation depth - the length or number of compositions in the continuation. For example the depth of top is zero and the depth of $\gamma \circ \text{appc}(v)$ is one plus the depth of γ .

Lemma (max.stack): For computations that do not involve context switching the maximum stack depth is a derived property.

Proof (max.stack): Let D be the primitive derivation over pairs of integers defined by

$$D(\hat{r}) = \begin{cases} \lambda x. \text{let}\{m \leftarrow \text{Addl}(\text{Car}(x))\} \text{Cons}[m, \text{Max}[m, \text{Cdr}(x)]] \\ \quad \text{if } r \in \{\text{if}, \text{app}, \text{cart}, \text{fst}, \text{rst}\} \\ \lambda x. \text{Cons}[\text{Sub1}(\text{Car}(x)), \text{Cdr}(x)] \\ \quad \text{if } r \in \{\text{ifi}, \text{cartc}, \text{fstc}, \text{rstc}, \text{pfn}, \circ \mid o \in O^D\} \\ | \quad \text{otherwise} \end{cases}$$

where $\text{Max}[x, y]$ is the maximum of x and y for integers x, y . If sw is not used in Σ and m is the length of the continuation component of 1^{st} (C) then by induction on the length of Σ we see that $\text{Car}(\Sigma^D(\text{Cons}[m, m]))$ is the length of the continuation component of the last state in Σ and $\text{Cdr}(\Sigma^D(\text{Cons}[m, m]))$ is the maximum of the continuation lengths for states occurring in Σ . \square max.stack

Corollary (no.max.stack): Maximum stack depth is not an s.d.p. for arbitrary computations.

Proof (no.max.stack): This follows from the universality of trace and the fact that maximum continuation length is not computable from the trace. To see the latter, note that $\langle \text{sw} \rangle$ is the trace of computation sequences with arbitrarily long continuations. \square no.m.stack

7.3. Derived programs

Rather than use derived properties directly to treat intensional properties of programs, we will define a derivation map and show how this can be used to convert intensional properties of programs into extensional properties of derived programs.

Definition (Derivation map): Let $\mathcal{D}\mathcal{S}$ be a new symbol. A derivation map is a family of functions A on Rum domains

$$\Delta \in [\mathbb{F} \rightarrow \mathbb{F}] \oplus [\mathbb{F}_{cs} \rightarrow \mathbb{F}] \oplus [\mathbb{C}_o \rightarrow \mathbb{C}_o] \oplus [\mathbb{V}^* \rightarrow \mathbb{V}^*] \oplus [\mathbb{D}_t \times \mathbb{V} \rightarrow \mathbb{D}_t] \oplus [\mathbb{S}_t \times \mathbb{V} \rightarrow \mathbb{S}_t]$$

such that (writing χ^D for $\Delta(\chi)$ to emphasize the dependence on the primitive derivation D)

$$\begin{aligned}
 d^D &= d \\
 o^D &= \lambda[ds, x][D(\partial ds), o(s)] \\
 (\langle \lambda x. \varphi : \xi \rangle \in V)^D &= (\lambda[ds, x](\varphi^D) : \xi^D) \\
 (\gamma \in V)^D &= (\lambda[ds, x]c[D(\widehat{sw}ds), x] : c \leftarrow \gamma^D) \\
 [a_1, \dots, a_n]^D &= [a_1^D, \dots, a_n^D] \\
 \langle \varphi : \xi \rangle^D(a) &= \langle \varphi^D : \xi^D \{ds \leftarrow a\} \rangle \\
 \xi^D(x) &= (\xi(x))^D \quad ;; \text{ for any variable } x \text{ in the domain of } \xi \\
 \text{top}^D &= \text{top} \\
 (\gamma \circ \langle \varphi_{cs} : \xi \rangle)^D &= \gamma^D \circ (\langle \varphi_{cs} \in F_{cs} \rangle^D : \xi^D) \\
 (\gamma \triangleright \delta)^D(a) &= \gamma^D \triangleright (\delta^D(a)) \\
 (\gamma \triangleleft v)^D(a) &= \gamma^D \triangleleft [a, v^D]
 \end{aligned}$$

and

$$\zeta_0 \xrightarrow{\Sigma} \zeta_1 \Rightarrow \zeta_0^D(a) \rightarrowtail \zeta_1^D(\Sigma^D(a))$$

Here as in other situations where one is defining a family of functions on *Rum* domains it is necessary to distinguish between $\langle \lambda x. \varphi : \xi \rangle$ the dtree and $\langle \lambda x. \varphi : \xi \rangle$ the pfn. To make this distinction we write $\langle \lambda x. \varphi : \xi \rangle \in V$ for the pfn. For continuations we write $\gamma \in V$ for γ as an element of the computation domain and simply γ as a component of a computation state. We write $\varphi \in F_{cs}$ when we are viewing φ as a continuation segment form rather than simply as a form.

The key fact about derivation maps is that derived pfns compute derived property functions. This is expressed by the theorem (**der.map**).

Theorem (der.map): For any $A \subset V$, D a primitive derivation over A and $a \in A$

$$p(v) \in V^* \Rightarrow \text{fst}(p^D[a, v^D]) \cong Cs(p(v))^D(a).$$

Note. An important property of *Rum* computation is that if $p(v) \in V^*$ then for any continuation γ there is a value u and a computation sequence $\Sigma \gamma \triangleright p(v) \xrightarrow{\Sigma} \gamma \triangleleft u$ and Σ^D is independent of the choice of γ . **Endnote.**

For the remainder of this section we assume that D is a constant symbol whose interpretation is a primitive derivation and that the new symbol ∂s ranges over V .

Theorem (dermap.exists): There exists a family of functions A that is a derivation map.

Proof (dermap.exists): From the definition it is easy to see that a derivation map is determined by its action on forms and continuation segments. Figure 9 defines such an action. The theory would be a little simpler if we worked in basic Rum with no constant symbols. However the practice is much simpler if we develop the theory accounting for the distinction between constant and variable symbols. Thus for each constant symbol s we choose a new constant symbol s' (the derived symbol associated with s) whose interpretation is the derivation of the interpretation of s . For variable symbols we take s' to be s . To check that the map defined by Figure 9 is indeed a derivation map one only needs to check the step requirement for each step rule. We leave this as an exercise. In fact the action of the derivation map on forms was obtained by looking at what was needed to prove the step requirement. \square

dermap.exists

To facilitate working with derived programs we have a number of lemmas for common special cases. To simplify matters we will assume in the remainder of this section that D is a primitive derivation over the integers for counting data operations. Thus for $r \in \text{Rules}$

$$D(r) \begin{cases} \tilde{\in} \{\text{I}, \text{Addl}\} & \text{if } r = \hat{o} \text{ for } o \in \mathbb{O}^D \\ \cong \text{I} & \text{otherwise.} \end{cases}$$

The general case is similar, just messier. Under these assumptions we have the following facts about the derivation map.

Lemma (dop.der): For data operations o

- (app) $(\varphi_0(\varphi_1))^D \cong \text{let}\{[ds, f_*] \leftarrow \varphi_0^D\} \text{let}\{[ds, x_*] \leftarrow \varphi_1^D\} f_*[ds, x_*]$
- (cart) $[\varphi_0, \varphi_1]^D \cong \text{let}\{[ds, x_*] \leftarrow \varphi_0^D\} \text{let}\{ds, y_* \leftarrow \varphi_1^D\} [ds, x_*, y_*]$
- (s.app) $(s(\varphi))^D \cong s'(\varphi^D)$
- (o.app) $o(x)^D \cong [D(\hat{o}ds), o(x)] \quad ;; o \in \mathbb{O}^D$
- (if.dop) $D(\hat{o}) = \text{I} \Rightarrow \text{if}(o(x), \varphi_1, \varphi_2)^D \cong \text{if}(o(x), \varphi_1^D, \varphi_2^D)$

Proof (dop.der): The proofs are essentially computation using the definition of the derivation map A , the simplifying assumptions (**letv**), (**let.perm**) and the cart laws. \square

The key to computing derived properties of recursively defined pfns is the derived recursion lemma (**der.rec**) and its corollaries. The derived recursion lemma says that the derivation of a pfn defined by a recursion equation is a pfn defined by

$$\begin{aligned}
s^D &= [D(\widehat{\text{sym}}, ds), s'] \\
m\ t^D &= [D(\widehat{\text{mt}}, ds), \text{mt}] \\
\text{top}^D &= [D(\widehat{\text{top}}, ds), \lambda[ds, x]\text{top}[D(\widehat{\text{sw}}, ds), x]] \\
(\lambda x.\varphi)^D &= [D(\widehat{\text{lam}}, ds), \lambda[ds, x]\varphi^D] \\
(\varphi_0(\varphi_1))^D &= \text{let}\{ds \leftarrow D(\widehat{\text{app}}, ds)\}(\text{appi}(\varphi_1)^D)(\varphi_0^D) \\
\text{appi}(\varphi_1)^D &= \lambda[ds, f_*]\text{let}\{ds \leftarrow D(\widehat{\text{appi}}, ds)\}(\text{appc}(f_*)^D)(\varphi_1^D) \\
\text{appc}(f_*)^D &= \lambda[ds, x_*]f_*[D(\widehat{\text{appc}}, ds), x_*] \\
\text{if}(\varphi_0, \varphi_1, \varphi_2)^D &= \text{let}\{ds \leftarrow D(\widehat{\text{if}}, ds)\}(\text{ifi}(\varphi_1, \varphi_2)^D)(\varphi_0^D) \\
\text{ifi}(\varphi_1, \varphi_2)^D &= \lambda[ds, x_*]\text{let}\{ds \leftarrow D(\widehat{\text{ifi}}, ds)\}\text{if}(x_*, \varphi_1^D, \varphi_2^D) \\
[\varphi_0, \varphi_1]^D &= \text{let}\{ds \leftarrow D(\widehat{\text{cart}}, ds)\}(\text{carti}(\varphi_1)^D)(\varphi_0^D) \\
\text{carti}(\varphi_1)^D &= \lambda[ds, x_*]\text{let}\{ds \leftarrow D(\widehat{\text{carti}}, ds)\}(\text{cartc}(x_*)^D)(\varphi_1^D) \\
\text{cartc}(x_*)^D &= \lambda[ds, y_*].[D(\widehat{\text{cartc}}, ds), x_*, y_*] \\
\text{fst}(\varphi)^D &= \text{let}\{\& \leftarrow D(\widehat{\text{fst}}, ds)\}\text{fstc}^D(\varphi^D) \\
\text{fstc}^D &= \lambda[ds, x_*].[D(\widehat{\text{fstc}}, ds), \text{fst}(x_*)] \\
\text{rst}(\varphi)^D &= \text{let}\{\& \leftarrow D(\widehat{\text{rst}}, ds)\}\text{rstc}^D(\varphi^D) \\
\text{rstc}^D &= \lambda[ds, x_*].[D(\widehat{\text{rstc}}, ds), \text{rst}(x_*)] \\
(\text{note}(c)\varphi)^D &= \text{note}(c)\text{let}\{c \leftarrow \lambda[ds, x_*].c[D(\widehat{\text{sw}}, ds), x_*]\}\text{let}\{ds \leftarrow D(\widehat{\text{note}}, ds)\}\varphi^D
\end{aligned}$$

where f_*, x_*, y_* are chosen fresh

Figure 9. Derivation map action on forms

a corresponding derived recursion equation. The corollaries extend this to multiary pfns.

Lemma (der.rec): Let F be a constant symbol defined by

$$F(x) \leftarrow \varphi$$

then the derivation F' of F is defined (mod \cong) by

$$F'[ds, x] \leftarrow \varphi^D$$

Corollary (der.rec.1): If G is a constant symbol defined by

$$G(x_1, x_2) \leftarrow \varphi$$

then the derivation G' of G is defined (mod \cong) by

$$G'[ds, x] \leftarrow [ds, G_1(x)]$$

$$G_1(x_1)[ds, x_2] \leftarrow \varphi^D$$

Corollary (der.rec.2): For G, G', G_1 as in (der.rec.1)

$$G(\varphi_1, \varphi_2)^D \cong \text{let}\{[ds, x_1] \leftarrow \varphi_1^D\} \text{let}\{[ds, x_2] \leftarrow \varphi_2^D\} G_1(x_1)[ds, x_2]$$

Again the proofs are just straightforward (but tedious) computations simplified somewhat by (dop.der). As an example we prove (der.rec.2).

Proof (der.rec.2):

$$\begin{aligned} & G(\varphi_1, \varphi_2)^D \\ & \cong \text{let}\{[ds, g] \leftarrow \text{let}\{[ds_1, x_1] \leftarrow \varphi_1^D\} G'[ds_1, x_1]\} \\ & \quad \text{let}\{[ds, x_2] \leftarrow \varphi_2^D\} \\ & \quad g[ds, x_2] \\ & \quad \text{;; using (dop.der.s.app), recall } G(\varphi_1, \varphi_2) \text{ abbreviates } (G(\varphi_1))(\varphi_2) \\ & \cong \text{let}\{[ds, g] \leftarrow \text{let}\{[ds_1, x_1] \leftarrow \varphi_1^D\} [ds_1, G_1(x_1)]\} \quad \text{;; (der.rec.1)} \\ & \quad \text{let}\{[ds, x_2] \leftarrow \varphi_2^D\} \\ & \quad g[ds, x_2] \\ & \cong \text{let}\{[ds_1, x_1] \leftarrow \varphi_1^D\} \quad \text{;; (let.perm)} \\ & \quad \text{let}\{[ds, g] \leftarrow [ds_1, G_1(x_1)]\} \\ & \quad \text{let}\{[ds, x_2] \leftarrow \varphi_2^D\} \\ & \quad g[ds, x_2] \\ & \cong \text{let}\{[ds, x_1] \leftarrow \varphi_1^D\} \text{let}\{[ds, x_2] \leftarrow \varphi_2^D\} G_1(x_1)[ds, x_2] \quad \text{;; (letv)} \end{aligned}$$

□ der.rec.2

7.4. Analysis of tree product computations.

To see how derived properties and programs can be used to analyze intensional properties of programs, we return to the tree product computations. In the following we assume 0 is a subset of {Car, Cdr, *} and D is the primitive derivation sequence defining count (0).

7.4.1. Analysis of the recursive tree product pfn

We begin with the simple recursive pfn **Tp**. Recall

$$\text{D} \quad \text{Tp}(x) \leftarrow \text{if}(\text{Atom}(x), x, *[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))])$$

To analyze properties of **Tp** we define the pfns **Cells** and **Nodes** where for number trees x , **Cells(x)** is the number of *conses* used to construct x , and **Nodes(x)** is the number of nodes excluding the root.

$$\text{D} \quad \text{Cells}(x) \leftarrow \text{if}(\text{Atom}(x), 0, 1 + \text{Cells}(\text{Car}(x)) + \text{Cells}(\text{Cdr}(x)))$$

$$\text{D} \quad \text{Nodes}(x) \leftarrow \text{if}(\text{Atom}(x), 0, 2 + \text{Nodes}(\text{Car}(x)) + \text{Nodes}(\text{Cdr}(x)))$$

The facts about the computation of **Tp** given in §2 are formalized by the theorem **(Tp.cnt)**.¹

Theorem (Tp.cnt):

$$(\text{mult}) \quad \text{count}(\{*\}, \text{Tp}(x)) \cong \text{Cells}(x)$$

$$(\text{ad}) \quad \text{count}(\{\text{Car}, \text{Cdr}\}, \text{Tp}(x)) \cong \text{Nodes}(x)$$

Proof (Tp.cnt): Define

$$\text{D} \quad \text{TpCnt}(x) \leftarrow \text{fst}(\text{Tp}'[0, x])$$

then by the derivation theorem we need only show

$$0 = \{\text{Car}, \text{Cdr}\} \Rightarrow \text{TpCnt}(x) \cong \text{Nodes}(x)$$

$$0 = \{*\} \Rightarrow \text{TpCnt}(x) \cong \text{Cells}(x)$$

This follows by an easy induction on number trees from the lemma **(tpcnt)**.

$\square \text{Tp.cnt}$

Lemma (tpcnt):

$$\text{TpCnt}(x) \cong \text{if}(\text{Atom}(x), 0, \text{Dcnt}(\text{TpCnt}(\text{Car}(x))) + \text{TpCnt}(\text{Cdr}(x)))$$

where

$$\text{D} \quad \text{Dcnt}(x) \leftarrow \text{D}(*, \text{D}(\widehat{\text{Cdr}}, (\text{D}(\widehat{\text{Car}}, x))))$$

¹ Note that the expression **Tp(x)** denotes a dtree in expressions such as $\text{count}(\{*\}, \text{Tp}(x))$.

To prove (tpcnt) we first prove two lemmas about the derivation Tp' of Tp .

Lemma (tpd.eq):

$$\begin{aligned} Tp'[ds, x] \cong & \text{if}(\text{Atom}(x), \\ & [ds, x], \\ & \text{let}\{[ds, x,] \leftarrow Tp'[\widehat{D(\text{Car}, ds)}, \text{Car}(x)]\} \\ & \text{let}\{[ds, x_d] \leftarrow Tp'[\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)]\} \\ & [D(\widehat{*}, ds), *[x_a, x_d]]) \end{aligned}$$

Lemma (tpd.sum):

$$Tp'[ds, x] \cong \text{let}\{[d, y] \leftarrow Tp'[0, x]\}[ds + d, y]$$

Proof (tpd.eq): By assumption we have $D(r) = I$ for r in Rules – {Car, Cdr, *}. Thus

$$\begin{aligned} (\text{L a}) \quad & \text{Car}(x)^D \cong [\widehat{D(\text{Car}, ds)}, \text{Car}(x)] \quad \text{;; (dop.der.o.app)} \\ (\text{L b}) \quad & \text{Cdr}(x)^D \cong [\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)] \quad \text{;; (dop.der.o.app)} \\ (1.\text{c}) \quad & *[\varphi_0, \varphi_1]^D \cong \text{let}\{[ds, x_a] \leftarrow \varphi_0^D\} \text{let}\{[ds, a] \leftarrow \varphi_1^D\} [D(\widehat{*}, ds), *[x_a, x_d]] \\ & \text{;; (dop.der.cart), (dop.der.s.app), defn } *' \\ (1.\text{d}) \quad & Tp(\text{Car}(x))^D \cong Tp'[\widehat{D(\text{Car}, ds)}, \text{Car}(x)] \quad \text{;; (dop.der.s.app), (1.a)} \\ (1.\text{e}) \quad & Tp(\text{Cdr}(x))^D \cong Tp'[\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)] \quad \text{;; (dop.der.s.app), (Lb)} \\ (1.\text{f}) \quad & *[Tp(\text{Car}(x)), Tp(\text{Cdr}(x))]^D \cong \text{let}\{[ds, x_a] \leftarrow Tp'[\widehat{D(\text{Car}, ds)}, \text{Car}(x)]\} \\ & \quad \text{let}\{[ds, x_d] \leftarrow Tp'[\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)]\} \\ & \quad [D(\widehat{*}, ds), *[x_a, x_d]] \\ & \text{;; (1.c,d,e)} \end{aligned}$$

$$(1.g) \quad \text{Tp}'[ds, x] \cong \text{if}(\text{Atom}(x), [ds, x], *[\text{Tp}(\text{Car}(x)), \text{Tp}(\text{Cdr}(x))]^D)$$

;; (**der.rec**) applied to **Tp**

$\cong \text{if}(\text{Atom}(x),$

$[ds, x],$

$\text{let}\{[ds, x_a] \leftarrow \text{Tp}'[\widehat{D(\text{Car}, ds)}, \text{Car}(x)]\}$

$\text{let}\{[ds, x_d] \leftarrow \text{Tp}'[\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)]\}$

$[\widehat{D(*, ds)}, *[x_a, x_d]])$

;; (**if.op**), (1.f)

$\square_{\text{tpd.eq}}$

Proof (tpd.sum): by induction on number trees

Case (Atom(x)):

$$\text{Tp}'[ds, x] \cong [ds, x] \quad ;; (\text{tpd.eq})$$

$\cong \text{let}\{[d, y] \leftarrow [0, x]\}[ds + d, y]) \quad ;; (\text{letv}) \text{ and arithmetic}$

$\cong \text{let}\{[d, y] \leftarrow \text{Tp}'[0, x]\}[ds + d, y]) \quad ;; (\text{tpd.eq})$

Case ($\neg \text{Atom}(x)$):

$$\text{Tp}'[ds, x] \cong \text{let}\{[ds, xx] \leftarrow \text{Tp}'[\widehat{D(\text{Car}, ds)}, \text{Car}(x)]\}$$

$\text{let}\{[ds, yy] \leftarrow \text{Tp}'[\widehat{D(\text{Cdr}, ds)}, \text{Cdr}(x)]\}$

$[\widehat{D(*, ds)}, *[xx, yy]]) \quad ;; (\text{tpd.eq})$

$\cong \text{let}\{[ds, xx] \leftarrow \text{let}\{[d_a, x_a] \leftarrow \text{Tp}'[0, \text{Car}(x)]\}[\widehat{D(\text{Car}, ds)} + d_a, x_a]\}$

$\text{let}\{[ds, xx] \leftarrow \text{let}\{[d_d, x_d] \leftarrow \text{Tp}'[0, \text{Cdr}(x)]\}[\widehat{D(\text{Cdr}, ds)} + d_d, x_d]\}$

$[\widehat{D(*, ds)}, *[xx, yy]]) \quad ;; \text{induction hypothesis}$

$\cong \text{let}\{[d, , x,] \leftarrow \text{Tp}'[0, \text{Car}(x)]\} \text{let}\{[ds, xx] \leftarrow [\widehat{D(\text{Car}, ds)} + d_a, x_a]\}$

$\text{let}\{[d_d, x_d] \leftarrow \text{Tp}'[0, \text{Cdr}(x)]\} \text{let}\{[ds, yy] \leftarrow [\widehat{D(\text{Cdr}, ds)} + d_d, x_d]\}$

$[\widehat{D(*, ds)}, *[xx, yy]]) \quad ;; \text{let.perm}$

$\cong \text{let}\{[d_a, x_a] \leftarrow \text{Tp}'[0, \text{Car}(x)]\}$

$\text{let}\{[d_d, x_d] \leftarrow \text{Tp}'[0, \text{Cdr}(x)]\}$

$[ds + \text{Dcnt}(d_a + d_d), *[x_a, y_a]])$

$;; (\text{letv}), \text{defn Dcnt, arithmetic}$

and

```

let{[d, YI ← Tp'[0,x]}[ds + d, YI
  ≅ let{ [d, y] ←
    let{ [d0, xx] ← Tp'[D(Car, 0), Car(x)]}
    let{ [do, yy] ← Tp'[D(Cdr, d0), Cdr(x)]}
      [D(*, do), *[xx, yy]]}
    [ds + d, YI      ;; (tpd.eq)

  ≅ let{ [d0, xx] ← Tp'[D(Car, 0), Car(x)]}
    let{ [d0, yy] ← Tp'[D(Cdr, do), Cdr(x)]}
      [ds + D(*, do), *[xx, yy]]      ;; (let.perm,letv)

  ≅ let{ [do, xx] ← let{[da, xa] ← Tp'[0, Car(x)]}[D(Car, 0) + da, xa]}
    let{[d0, yy] ← let{[dd, xd] ← Tp'[0, Cdr(x)]}[D(Cdr, do) + dd, xd]}
      [ds + D(*, do), *[xx, yy]]      ;; induction hypothesis

  ≅ let{[da, xa] ← Tp'[0, Car(x)]}
    let{[dd, xd] ← Tp'[0, Cdr(x)]}
      [ds + Dcnt(da + dd), *[xa, xd]]
        ;; (let.perm,letv), defn Dcnt, arithmetic

```

$\square_{\text{tpd.sum}}$

Proof (tpcnt): This is an easy induction on number trees.

```

TpCnt(x) ≈ if(Atom(x),
                 fst[0,x],
                 fst( let{ [ds , xx] ← Tp'[D(Car,0), Car(x)]}
                      let{ [ds,yy] ← Tp'[D(Cdr,ds), Cdr(x)]}
                      [D(*,ds),*[xx,yy]]))
                 ;; defn TpCnt, (tpd.eq), (dist)

≈ if(Atom(x),
      0,
      let{ds ← fst(Tp'[D(Car,0), Car(x)])}
      let{ds←fst(Tp'[D(Cdr,ds) , Cdr(x)])}
      D(*,ds))
      ;; fst laws

≈ if(Atom(x),
      0,
      let{ds ← let{ds1 ← fst(Tp'[0,Car(x)])}D(Car,0) + ds1)}
      let{ds←let{ds2←fst(Tp'[0,Cdr(x)])}D(Cdr,ds) + ds2)}
      D(*,ds))
      ;; fst laws,(tpd.sum)

≈ if(Atom(x),
      0,
      let{ds ← D(Car,0) + TpCnt(Car(x))})
      let{ds←D(Cdr,ds) + TpCnt(Cdr(x))}}
      D(*,ds))
      ;; (let.perm,letv), defn TpCnt

≈ if(Atom(x), 0, Dcnt(TpCnt(Car(x))) + TpCnt(Cdr(x)))
      ;; arithmetic

```

□_{tpcnt}

7.4.2. Analysis of the escaping tree product pfn

The analysis of **Tps** is similar to that of **Tp**. Below we outline the main points. First recall that **Tps** is defined by

- D $\text{Tps}(x) \leftarrow \text{note}(c)\text{Ts}(x, c)$
- D $\text{Ts}(x, c) \leftarrow \text{if}(\text{Atom}(x), \text{if}(\text{Zerop}(x), c(0), x), \text{Ts}(\text{Car}(x), c) * \text{Ts}(\text{Cdr}(x), c))$

To analyze properties of **Tp** we define the pfns **CellsB** and **NodesB** where for number trees x , $\text{CellsB}(x)$ is the number of cells whose subtree occurs before the first zero in a depth-first traversal of x , and $\text{NodesB}(x)$ is the number of nodes visited before discovering the first zero. **CellsB** and **NodesB** are defined by

- D $\text{CellsB}(x) \leftarrow \text{if}(\text{Atom}(x),$
 $0,$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $\text{CellsB}(\text{Car}(x)),$
 $\text{if}(\text{Inz}(\text{Cdr}(x)),$
 $\text{CellsB}(\text{Car}(x)) + \text{CellsB}(\text{Cdr}(x)),$
 $1 + \text{CellsB}(\text{Car}(x)) + \text{CellsB}(\text{Cdr}(x))))$
- D $\text{NodesB}(x) \leftarrow \text{if}(\text{Atom}(x),$
 $0,$
 $\text{if}(\text{Inz}(\text{Car}(x)),$
 $1 + \text{NodesB}(\text{Car}(x)),$
 $2 + \text{NodesB}(\text{Car}(x)) + \text{NodesB}(\text{Cdr}(x))))$

The facts about the computation of **Tps** given in §2 are formalized by the theorem (Tps.cnt).

Theorem (Tps.cnt):

- (mult) $\text{count}(\{\ast\}, \text{Tps}(x)) = \text{CellsB}(x)$
- (ad) $\text{count}(\{\text{Car}, \text{Cdr}\}, \text{Tps}(x)) = \text{NodesB}(x)$

Proof (Tps.cnt): Define

- D $\text{TpsCnt}(x) \leftarrow \text{fst}(\text{Tps}'[0, x])$

then by the derivation theorem we need only show

$$(\text{TpS}\text{Cnt}.\text{mult}) \quad \mathbf{0} = \{\ast\} \Rightarrow \text{TpS}\text{Cnt}(x) \cong \text{CellsB}(x)$$

$$(\text{TpS}\text{Cnt}.\text{ad}) \quad \mathbf{0} = \{\text{Car}, \text{Cdr}\} \Rightarrow \text{TpS}\text{Cnt}(x) \cong \text{NodesB}(x)$$

For this we define two auxiliary pfns TsD and $\mathbf{B4}$.

$$\begin{aligned} \mathbf{D} \quad \text{TsD}(x)[ds, c] &\leftarrow \text{if}(\text{Atom}(x), \\ &\quad \text{if}(\text{Zerop}(x), c(x), x), \\ &\quad *[\text{Ts}(\text{Car}(x), c), \text{Ts}(\text{Cdr}(x), c)])^{\mathbf{D}} \\ \mathbf{D} \quad \mathbf{B4}(x, d) &\leftarrow \text{if}(\text{Atom}(x), \\ &\quad d, \\ &\quad \text{if}(\text{Inz}(\text{Car}(x)), \\ &\quad \widehat{\mathbf{B4}(\text{Car}(x)\mathbf{D}(\widehat{\text{Car}}d))}), \\ &\quad \text{if}(\text{Inz}(\text{Cdr}(x)), \\ &\quad \widehat{\mathbf{B4}(\text{Cdr}(x)\mathbf{D}(\widehat{\text{Cdr}}\mathbf{B4}(\text{Car}(x)\mathbf{D}(\widehat{\text{Car}}d)))))}, \\ &\quad \mathbf{D}(*\widehat{\mathbf{B4}(\text{Cdr}(x)\mathbf{D}(\widehat{\text{Cdr}}\mathbf{B4}(\text{Car}(x)\mathbf{D}(\widehat{\text{Car}}d)))))})) \end{aligned}$$

By the derived recursion theorem and assumptions on \mathbf{D} we have

$$\text{TpS}'[ds, x] \cong (\text{note}(c)\text{Ts}(x, c))^{\mathbf{D}} \cong \text{note}(c)(\text{Ts}(x, c))^{\mathbf{D}} \cong \text{note}(c)\text{TsD}(x)[ds, c])$$

$\mathbf{B4}$ is a counter that generalizes CellsB and NodesB and we have the following

Lemma ($\text{TsD}.\text{cnt}$): For $x \in NTree$, $d \in \mathbb{D}_{\text{Int}}$, $c \in \mathbb{C}_o$

$$(0) \quad \mathbf{0} = \{\text{Car}, \text{Cdr}\} \Rightarrow \mathbf{B4}(x, d) \cong \text{NodesB}(x) + d$$

$$\mathbf{0} = \{\ast\} \Rightarrow \mathbf{B4}(x, d) \cong \text{CellsB}(x) + d$$

$$(1) \quad \text{TsD}(x)[d, c] \cong \text{if}(\text{Inz}(x), c[\mathbf{B4}(x, d), 0], [\mathbf{B4}(x, d), \text{Tp}(x)])$$

$$(2) \quad \text{TpS}'[d, x] \cong [\mathbf{B4}(x, d), \text{Tp}(x)]$$

The proof of (0) is an ordinary induction on number trees. The proofs of (1,2) follow the outline of the proofs of $(\text{Ts}.\text{Tp})$ (§6) and $(\text{TpS}.\text{Tp})$ (§2). $(\text{TpS}\text{Cnt}.\text{mult})$ and $(\text{TpS}\text{Cnt}.\text{ad})$ follow easily from $(\text{TsD}.\text{cnt})$. $\square_{\text{TpS}.\text{cnt}}$

7.5. Possible elaborations

The notion of simple derived property accounts for many of the intensional properties of computations of interest. There are several possible elaborations that would allow even more to be expressed.

First note that we cheated slightly in saying that the maximum stack depth is an s.d.p. Namely it is **Cdr** of the s.d.p. defined in the proof of (**max.stack**). To fix this we generalize s.d.p.s to be given by a primitive derivation \mathbf{D} , an initial value a , and an output function p . Then the given derived property of a computation sequence Σ is $p(\Sigma^{\mathbf{D}}(a))$.

We can also generalize the class of primitive derivations. One possibility is for primitive derivations to take an encoding of the computation state as an argument (from which the rule applied can be deduced). This would allow a more complete trace to be defined which includes for example not only the data operation names but also the arguments. Also deriving functions for data operations could make the cost of the operation depend on the argument.

Another possibility is to add more structure to objects such as forms and pfns. For example one could add tags to λ s and to data operations so that different occurrences could be distinguished. This trick was used by [Wegbreit 1976] in an intensional analysis of programs.

8. Conclusions

We have developed an intensional semantic theory of function and control abstractions and illustrated the use of this theory for programming and for proving both intensional and extensional properties of programs.

The extensional theory is based on the notions of **operational** approximation, equivalence, and membership. Operational approximation **is a partial** ordering that abstracts the “less defined than” relation and has many of the properties of the partial ordering in the graph model of the lambda calculus [Scott 1976], including extensionality, restricted eta, and the least-fixed point property of the recursion operator.¹ The operational membership relation allows one to express definedness and descriptive type information. These relations together with mechanisms for defining subsets of the value domain are the basis for a rich theory for expressing properties of programs. The theory presented is first-order except for the reliance on semantic definitions of sets of values. This can be fixed by taking the approach of [Feferman 1975,79,85] to develop a first-order theory of operations and classes over a computational universe.

The intensional theory is based on the notion of derived property. Derived properties of programs can be obtained by “instrumenting” the operational semantics. Computation of derived properties of programs can be thought of as a “non-standard” interpretation of programs. Programs can be transformed uniformly to compute derived properties and hence reduce reasoning about derived properties to reasoning about ordinary programs. We gave a systematic transformation for obtaining derived programs based on a particular form of definition of derived property. More generally, one could obtain derived programs by specializing an instrumented interpreter, and a transformation from programs to corresponding derived programs could then be obtained by specializing the specializer. These are examples of a general technique known as partial evaluation (cf. [Jones, et. al. 1989]).

Although we obtained our extensional theory of function and control abstractions by starting with the *Rum* semantic theory and extracting the key laws as axioms one could also take the view that these axioms characterize a class of models (*Rum* being one such model) and investigate this class of models in more depth. Such an investigation would build on the work of [Moggi 1986, 1989]. Also, the operational semantics and derived properties or other non-standard interpretations can be considered as models of a common semantic language sharing some basic axioms. Thus we can structure the variety of interpretations of programs in

¹ For further discussion of the relation between operational approximation and the partial ordering induced by interpretation in Scott type domains see [Plotkin 1977].

an algebraic framework for programming language theories. Preliminary ideas for setting up such a framework are presented in [Talcott 1989].

An alternative approach to equational reasoning about function and control abstractions is given in [Felleisen and Friedman 1986, Felleisen 1987, Felleisen et al. 1987]. Here lambda-v-c equivalence is defined by **a two-level equational** calculus derived from an abstract machine similar to our state-transition semantics. Operational equivalence includes lambda-v-c equivalence and the rules generating lambda-v-c equivalence are provable in our **theory**.

Reduction calculi and operational approximation **both provide a sound basis** for purely equational reasoning about programs. Calculi have the advantage that the reduction relations are inductively generated from primitive reductions (such as beta-conversion) by closure operations (such as transitive closure or congruence closure). Equations proved in a calculus continue to hold when the language is extended to treat additional language constructs. Operational approximation is, by definition, sensitive to the set of language constructs and basic data available. For example, in the call-by-value lambda calculus with only algebraic operations, a do-forever loop is equivalent to the totally undefined function for any value of its function parameter. But in the presence of control abstractions that provide a mechanism for escaping from or forgetting the current computation context, until loops can be defined using the do-forever loop (cf. [Talcott 85, Chapter V]). [Mason and Talcott 1989b] give examples of change in operational equivalence when memory and updating is introduced. Using operational approximation we can express and prove properties such as non-termination, computation induction, and least fixed-points, which cannot even be expressed in a reduction calculus framework. Finding the correct balance between insensitivity to extensions of the language and the expressive power of operational approximation is an important problem. Studying the laws of operational approximation and discovering natural extensions to reduction calculi provide useful insight into this problem.

In addition to function and control abstractions, Lisp- and Scheme-like languages also provide primitives for creating and manipulating objects with memory. [Mason and Talcott 1985] gives a semantics of a language of first order recursion equations acting on objects with memory (Lisp list structures) and gives many examples of proving program equivalence based directly on this semantics. [Mason 1986] builds on this work. A model-theoretic equivalence called *strong isomorphism* is introduced and used as the basis for studying program equivalence and transformational programming. Two expressions are strongly isomorphic if for any environment and memory they are either both undefined or both reduce to the same value and have the same effect on memory modulo production of garbage. Several decidability results are given, the laws of strong isomorphism are studied in some detail, and many examples of proving program properties are presented

including non-trivial programs such as the Robson copy algorithm and a structure editor that edits destructively (a copy of the original structure). [Felleisen 1987, 1988b] presents a two-level calculus for a scheme-like language with function, control, and assignment abstraction. In the basic language environments bind variables to mutable objects, variables can not be treated as values, and beta-value conversion is not a valid rule. Thus it is necessary to have two sorts of lambda variables – assignable and non-assignable. To account for memory and sharing the syntax is extended to include labeled values. A variety of examples of reasoning about programs are given. It was found necessary to extend the basic calculus in order to carry out some of the examples. [Mason and Talcott 1989b] gives an alternative approach to treating programs with memory and function abstractions. Here the call-by-value lambda calculus is extended by adding operations for creating, accessing, and updating memory cells. A syntactic representation of memory is developed that permits computation rules to be expressed by a reduction relation on expressions. This forms a natural basis for equational reasoning about objects with memory. The theory of operational approximation and equivalence for this case was studied and tools were developed for proving equivalence. In [Mason and Talcott 1989a] a formal system is presented for proving constrained equivalence for programs with side effects. Constrained equivalence is a relation between finite sets of constraints and a pair of expressions that holds in the case that the expressions are strongly isomorphic in all memory contexts satisfying the constraints. A constraint may require a variable to be an atom or cell, it may require a variable to be equal or distinct from another variable or constant, or it may require some component of the contents of a variable ranging over cells to be equal to some variable or constant. The deduction system is complete for first-order expressions that contain no occurrences of recursively defined functions and a decision procedure has been extracted from the proof of equivalence. In the first-order fragment strong isomorphism is the same as operational equivalence. In the full higher-order case strong isomorphism implies operational equivalence (but not conversely) and methods for proving strong isomorphism are useful for large classes of problems even in the higher-order case. [Mason and Talcott 1989c] gives many examples of applications of the theory developed in [Mason and Talcott 1989a,b] including extending first-order examples to the higher-order case, traversing structures for effect, relating objects and behavior descriptions, and memoizing thunks and streams. More complete surveys of reasoning about programs with memory can be found in [Mason 1986] and [Felleisen 1987].

9. References

Abelson, H. and G. J. Sussman

[1985] ***Structure and interpretation of computer programs***, (The MIT Press, McGraw-Hill Book Company).

Abramsky, S. and Hankin, C. (eds.)

[1987] ***Abstract interpretations of applicative languages*** (Michael Horwood, London)

Aczel, P.

[1977] An introduction to inductive definitions, in: ***Barwise 1977***, pp. 739-782.

Barendregt, H.

[1981] ***The lambda calculus: its syntax and semantics*** (North-Holland, Amsterdam).

Burge, W. H.

[1971] Some examples of the use of function producing functions, in: ***Proceedings, 2nd ACM symposium on symbolic and algebraic manipulation***, pp. 238-241.

[1975str] Stream processing functions, ***IBM journal of research and development***, 19, pp. 12-25.

[1975rec] Recursive ***programming techniques***, (Addison-Wesley).

Burstall, R. M.

[1968] Writing search algorithms in functional form, in: ***Machine intelligence 9***, edited by D. Michie, (Edinburgh University Press), pp. 373-385.

Conway, M.

[1963] Design of a separable transition-diagram compiler, Comm. **ACM**, 6, pp. 396-408.

Cousot, P. and Cousot, R.

[1977] Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixed points, in: ***4th ACM symposium on principles of programming languages*** pp. 238-252.

Danvy, O.

[1989] Abstracting Control (submitted for publication)

Feferman, S.

- [1975] A language and axioms for explicit mathematics, in: *Algebra and Logic*, Springer Lecture Notes in Mathematics, 450, pp.87-139.
- [1979] constructive theories of functions and classes, in *Logic Colloquium '78*, (North-Holland) pp. 159-224.
- [1982] Inductively presented systems and the formalization of meta-mathematics, in: *Logic colloquium* 80, edited by D. van Dalen, D. Lascar, and J. Smiley (North-Holland, Amsterdam) pp. 95-128.
- [1985] A Theory of Variable Types, *Revista Colombiana de Matemáticas*, 19 pp. 95–105.

Felleisen, M.

- [1987] *The calculus of lambda-v-CS conversion: A syntactic theory of control and state in imperative higher-order programming languages*, Ph.D. thesis, Indiana University.
- [1988a] The theory and practice of first-class prompts, *15th annual ACM Symposium on principles of programming languages*, pp. 180-190.
- [1988b] λ -v-CS: An extended X-calculus for Scheme, in: *Proceedings of the 1988 ACM conference on Lisp and functional programming*, pp. 72-85.

Felleisen, M. and Friedman, D. P.

- [1986] Control operators, the SECD-machine, and the X-calculus, in: *Proceedings of the conference on formal description of programming concepts part III. Ebberup Denmark, August 1986*.
- [1987] A calculus for assignments in higher-order languages, in: *Proc. 14th ACM symposium on principles of programming languages*, pp. 314-325.

Felleisen, M., Friedman, D. P., Kohlbecker E., and Druba B.

- [1987] A syntactic theory of sequential control, *Theoretical Computer Science*

Friedman, D. P. et.al.

- [1984] *Fundamental abstractions of programming languages*, Computer Science Department, Indiana University.

Goguen, J. A. and Meseguer, J.

- [1983] Initiality, induction, and computability, in: *Applications of algebra to language definitions and compilation*, edited by M. Nivat and J. Reynolds (Cambridge University Press).
- [1984] Equality, types, modules, and generics for logic programming, Center for the study of language and information, Stanford University, Report No. CSLI-84-5.

Henderson, P.

- [1980] *Functional programming: Application and implementation*, (Prentice-Hall).

Jones, N. D. and Mycroft, A.

- [1986] Dataflow analysis of applicative programs using minimal function graphs, in: *1 9th annual ACM symposium on principles of programming languages*, pp. 296-306.

Jones, N. D., Sestoft, P., and Søndergaard, H.

- [1989] Mix: A self-applicable partial evaluator for experiments in compiler generation, *Lisp and Symbolic Computation*, 2, pp. 9-50.

Kahn, G. and D. B. MacQueen

- [1977] Coroutines and networks of parallel processes, *Information processing 77* (North-Holland, Amsterdam) pp. 993-998.

Landin, P. J.

- [1964] The mechanical evaluation of expressions, *Computer journal*, 6, pp. 308-320.
- [1965] A correspondence between Algol 60 and Church's lambda notation, Comm. ACM, 8, pp. 89-101, 158-165.
- [1966] The next 700 programming languages, Comm. ACM, 9, pp. 157-166.

Mason, I. A.

- [1986] The semantics of destructive Lisp, Ph.D. Thesis, Stanford University. Also as CSLI Lecture Notes No. 5, Center for the Study of Language and Information, Stanford University.

Mason, I. A. and Talcott, C. L.

- [1989a] Axiomatizing operational equivalence in the presence of side effects. *Fourth annual symposium on logic in computer science*, (IEEE).
- [1989b] Programming, transforming, and proving with function abstractions and memories. *Proceedings of the 16th EATCS colloquium on automata, languages and programming, Stresa, Italy*.
- [1989c] Equivalence of programs and objects with memory: some examples, (in preparation).

Milne, R. and C. Strachey

- [1976] *A theory of programming language semantics* (Chapman and Hail, London).

Moggi, E.

- [1988] *The partial lambda-calculus*, Ph. D. thesis, University of Edinburgh.
- [1989] Computational lambda-calculus and monads, *Fourth annual symposium on logic in computer science*, (IEEE).

Morris, J. H.

- [1968] *Lambda calculus models of programming languages*, Ph.D. thesis, Massachusetts Institute of Technology.

Moschovakis Y. N.

- [1975] On the basic notions in the theory of induction, in: *Logic, foundations of mathematics, and computability theory: Proceedings of the 5th international congress of logic methodology and philosophy of science*, edited by R. E. Butts and J. Hintikka (D. Reidel, Boston) pp. 207-236.

Plotkin, G.

- [1975] Call-by-name, call-by-value and the lambda-v-calculus, *Theoretical Computer Science*, 1, pp. 125-159.
- [1977] LCF considered as a programming language, *Theoretical Computer Science*, 5. pp. 223-255. Queinnec, C. and Séniak, N. [1989] Puzzling with Current Puzzle *Lisp Pointers* 2 (3-4) pp. 4.

Rees, J., Clinger, W. (eds)

- [1986] The revised³ report on the algorithmic language Scheme, *Sigplan Notices*, 21(12), pp. 37-79.

Reynolds, J. C.

- [1972] Denotational interpreters for higher-order programming languages, in: ***Proceedings, A CM national convention***, pp. 717-740.

Scherlis, W. L.

- [1981] Program improvement by internal specialization, in: Conference ***record of the 8th annual ACM symposium on principles of programming languages***, pp. 4149.

Schmidt, D.A.

- [1986] ***Denotational semantics: a methodology for language development***, (Allyn and Bacon).

Scott, D.

- [1976] Data types as lattices, ***SIAM J. of Computing***, 5, pp. 522-587.

Scott, D. and C. Strachey

- [1971] Towards a mathematical semantics for computer languages, Oxford University Computing Laboratory, Technical Monograph PRG-6.

Steele, G. L. Jr.

- [1984] ***Common Lisp: the language*** (Digital Press).

Steele, G. L., and G. J. Sussman,

- [1975] Scheme, an interpreter for extended lambda calculus, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report 349.

Talcott, C.

- [1985] The essence of *Rum*: A theory of the intensional and extensional aspects of Lisp-type computation, Ph. D. Thesis, Stanford University.
- [1989] Algebraic methods in programming language theory (extended abstract), in: ***First International Conference on algebraic methodology and software technology, Iowa City***, Iowa. (full version in preparation).

Wegbreit , B.

- [1975] Mechanical program analysis, Comm. **ACM**, 18 , pp. 528-539.
- [1976] Goal directed program transformation, in: **Third ACM** symposium on principles *of programming languages*.

Wegner, P.

- [1971] Data structure models for programming languages, in: Proceedings *of a symposium on data structures in programming languages*, edited by J. Tou and P. Wegner, **SIGPLAN Notices**, 6 , pp. 1-54.
- [1972] The Vienna definition language, **Computing Surveys**, 4, pp. 5-63.

10. Appendix: Proofs and technical miscellany

This appendix collects proofs and discussions of technical issues that are relevant to the development of programming theories but not necessary for their use.

10.1. Properties of operational relations

We begin by defining a family of refinements of the operational relations. The facts operational stated in §4 are then restated and proved in more general form.

10.1.1. A refinement of operational approximation and equivalence

Operational equivalence as defined in §5 has the property that if two dtrees are equivalent then either both are undefined or both escape (returning a value to the top level) or both return a value to the calling context, independent of the context. In the case of value returned, the values are equivalent. However in the case of escaping, the values returned to the top level need not be equivalent. For example $\text{top}(\text{Bot}) \cong \text{top}(\text{l})$ both escape and return pfn's to the top level. But, Bot and l are clearly not equivalent values.

We can refine the operational approximation and equivalence relations to obtain relations \sqsubseteq_ω and \cong_ω that satisfy all the properties given for \sqsubseteq and \cong and, in addition, to have that related dtrees that escape return related values to the top level. The refinement is obtained by taking the limit of a chain of refinements of trivial approximation treating operational approximation as the first refinement of trivial approximation.

D (Refining operational relations): For $n \in \mathbb{N}$, χ a dtree, value or state, and $\alpha \leq w$ (the least infinite ordinal).

$$\zeta_0 \sqsubseteq_n \zeta_1 \stackrel{\text{df}}{\equiv} (\forall v_0)(\zeta_0 \rightarrowtail \text{top} \Delta v_0 \Rightarrow (\exists v_1)(v_0 \sqsubseteq_n v_1 \wedge \zeta_1 \rightarrowtail t \circ p \Delta v_1))$$

$$\delta_0 \sqsubseteq_{n+1} \delta_1 \stackrel{\text{df}}{\equiv} (\forall \gamma)(\gamma \triangleright \delta_0 \sqsubseteq_n \gamma \triangleright \delta_1)$$

$$v_0 \sqsubseteq_{n+1} v_1 \stackrel{\text{df}}{\equiv} (\forall \gamma)(\gamma \Delta v_0 \sqsubseteq_n \gamma \Delta v_1)$$

$$\chi_0 \sqsubseteq_\omega \chi_1 \stackrel{\text{df}}{\equiv} (\forall n)(\chi_0 \sqsubseteq_n \chi_1)$$

$$\chi_0 \cong_\alpha \chi_1 \stackrel{\text{df}}{\equiv} \chi_0 \sqsubseteq_\alpha \chi_1 \wedge \chi_1 \sqsubseteq_\alpha \chi_0$$

10.1.2. Proofs of easy consequences

Theorem (Pre-order): \sqsubseteq_m is reflexive and transitive for all m

Proof (Pre-order): An easy consequence of the definitions (using induction on m). \square

Theorem (Refinement): If $m \leq n$ then \sqsubseteq_n is a subrelation of \sqsubseteq_m .

Proof (Refinement): Exercise. \square

Theorem (Pointwise):

$$v_0 \sqsubseteq_{n+1} v_1 \Rightarrow |v_0| = |v_1| \wedge (\forall i < |v_0|)(v_0 \downarrow_i \sqsubseteq_{n+1} v_1 \downarrow_i)$$

Proof (Pointwise): Exercise. \square

Corollary (Pointwise): $\text{do } \sqsubseteq_m d_1 \Leftrightarrow d_0 = d_1$

Theorem (Computational facts):

$$(i) \quad \zeta_0 \succcurlyeq \zeta_1 \Rightarrow \zeta_0 \cong_m \zeta_1$$

$$(ii) \quad \zeta_0 \succcurlyeq \text{top} \Delta v_0 \wedge \zeta_1 \succcurlyeq \text{top} \Delta v_1 \Rightarrow \zeta_0 \sqsubseteq_m \zeta_1 \Leftrightarrow v_0 \sqsubseteq_m v_1$$

Proof (Computational facts): a direct consequence of the definitions and the unicity of the single-step relation and hence of evaluation when defined. \square

Theorem (Environment substitution): If $\xi_0(s) \sqsubseteq_{m+1} \xi_1(s)$ for all s free in φ then $\langle \varphi : \xi_0 \rangle \sqsubseteq_{m+1} \langle \varphi : \xi_1 \rangle$.

Proof (Environment substitution): This is an instance of the substitution theorem below. However it has a simple direct proof. Suppose φ has only one free variable x and $\xi_0(x) \sqsubseteq_{m+1} \xi_1$. Then for any continuation γ

$$\begin{aligned} \gamma \triangleright \langle \varphi : \xi_0 \rangle \\ \cong_m \gamma \circ \text{appc}(\langle \lambda x. \varphi : \{\} \rangle) \Delta v_0 &\quad ;; \text{ computation} \\ \sqsubseteq_m \gamma \circ \text{appc}(\langle \lambda x. \varphi : \{\} \rangle) \Delta v_1 &\quad ;; \text{ since } v_0 \sqsubseteq_{m+1} v_1 \\ \cong_m \gamma \triangleright \langle \varphi : \xi_1 \rangle &\quad ;; \text{ computation} \end{aligned}$$

The general case is an easy induction on the number of free variables in φ using the above argument. \square

10.1.3. Substitution

Recall that for any semantic entity (dtrees, values, continuations, and states) $\chi[]$ is an entity with holes of some sort (determined by context) and that then $\chi[\chi_0]$ is the result of filling the hole with χ_0 when χ_0 is an entity of the same sort as the holes. It is easy to see that the computation rules extend naturally to states with holes with steps uniformly parameterized by the holes except when **holes axe** actually touched. A continuation hole is only touched if it is the continuation component of a state, a dtree hole is only touched if it is the dtree component of a begin state, an operation hole is only touched if it is the value part of an appc continuation (see [Talcott 1985] V.3 for a similar elaboration).

Theorem (Substitution): If χ_0, χ_1 are dtrees, continuations, or values such that $\chi_0 \sqsubseteq_{n+1} \chi_1$ then $\chi[\chi_0] \sqsubseteq_{n+1} \chi[\chi_1]$ for χ any dtree or value with holes of a suitable sort.

Corollary (subst.omega): If χ_0, χ_1 are dtrees, continuations, or values such that $\chi_0 \sqsubseteq_\omega \chi_1$ then $\chi[\chi_0] \sqsubseteq_\omega \chi[\chi_1]$ for χ any dtree, value, or state with holes of a suitable sort.

Proof (Substitution): Let χ_0, χ_1 be continuations, dtrees, or operations. We will prove by induction on n that $\chi_0 \sqsubseteq_{n+1} \chi_1$ implies

- (i) $\zeta[\chi_0] \sqsubseteq_n \zeta[\chi_1]$ for any state $\zeta[]$ with holes of a suitable sort, and
- (ii) $\chi[\chi_0] \sqsubseteq_{n+1} \chi[\chi_1]$, for any dtree or value $\chi[]$ with holes of a suitable sort.

The case χ_0, χ_1 are arbitrary values follows easily.

Clearly, $u[\chi_0] \sqsubseteq_0 u[\chi_1]$ for any value $u[]$ with holes of a suitable sort. For each n , (ii) follows easily from (i). By the induction hypothesis, we need only show that (for any n) $\chi_0 \sqsubseteq_{n+1} \chi_1$ and $u[\chi_0] \sqsubseteq_n u[\chi_1]$ for any value $u[]$ with holes of a suitable sort implies $\zeta[\chi_0] \sqsubseteq_n \zeta[\chi_1]$ for any state $\zeta[]$ with holes of a suitable sort. This is done in the lemmas below: **(subst.c)** for the continuation case; **(subst.d)** for the dtree case; and **(subst.o)** for the operation case. \square

Lemma (subst .c): If

- (ret-hyp) $\gamma_0 \sqsubseteq_{m+1} \gamma_1$ (as values), and
- (top-hyp) $u[\gamma_0] \sqsubseteq_m u[\gamma_1]$ for all values $u[]$ with continuation holes
then $\zeta[\gamma_0] \sqsubseteq_m \zeta[\gamma_1]$ for all states $\zeta[]$ with continuation holes.

Proof (subst.c): By induction on the length of the computation of $\zeta[\gamma_0]$ (assuming defined) and cases on the form of $\zeta[]$.

Case (top): $\zeta[]$ has the form $\text{top} \triangle u[]$. Use (top-hyp). \square_{top}

Case (step-unif): $\zeta[\]$ has one of the forms

$$\begin{aligned} & \gamma[\] \circ \text{ifi}(\delta_1[\], \delta_2[\]) \triangleq u[\] \\ & \gamma[\] \circ \text{appi}(\delta[\]) \triangleq u[\] \\ & \gamma[\] \circ \text{appc}(\vartheta) \triangleq u[\] \\ & \gamma[\] \circ \text{carti}(\delta[\]) \triangleq u[\] \\ & \gamma[\] \circ \text{cartc}(v[\]) \triangleq u[\] \\ & \gamma[\] \circ \text{fstc} \triangleq u[\] \\ & \gamma[\] \circ \text{rstc} \triangleq u[\] \\ & \gamma[\] \triangleright \delta[\] \end{aligned}$$

Then there is a state ζ_0 such that $\zeta[\] \rightarrowtail \zeta_0[\]$ uniformly (with out touching the holes). By the computation induction hypothesis we have $\zeta_0[\gamma_0] \sqsubseteq_m \zeta_0[\gamma_1]$ and by the computation facts we are done. $\square_{\text{step-unif}}$

Case (hole): $\zeta[\]$ has the form $[\] \triangleq u[\]$. By the top case and uniform step arguments $\gamma_0 \triangleq u[\gamma_0] \sqsubseteq_m \gamma_0 \triangleq u[\gamma_1]$. By (ret-hyp) and computation

$$\gamma_0 \triangleq u[\gamma_1] \sqsubseteq_m \text{topappi}(u[\gamma_1]) \triangleq \gamma_0 \sqsubseteq_m \text{topappi}(u[\gamma_1]) \triangleq \gamma_1 \quad \text{cm } \gamma_1 \triangleq u[\gamma_1]$$

$\square_{\text{hole}} \square$

Lemma (subst.d): If

- (ret-hyp) $\delta_0 \sqsubseteq_{m+1} \delta_1$, and
- (top-hyp) $u[\delta_0] \sqsubseteq_m u[\delta_1]$ for all values $u[\]$ with dtree holes

then $\zeta[\delta_0] \sqsubseteq_m \zeta[\delta_1]$ for all states $\zeta[\]$ with dtree holes.

Proof (subst.d): The proof is analogous to the proof of (subst.c). The **only** difference is the hole case. In this case $\zeta[\]$ has the form $\gamma[\] \triangleright [\]$. By the uniform step argument $\gamma[\delta_0] \triangleright \delta_0 \sqsubseteq_m \gamma[\delta_1] \triangleright \delta_0$ and by (ret-hyp) $\gamma[\delta_1] \triangleright \delta_0 \sqsubseteq_m \gamma[\delta_1] \triangleright \delta_1$. \square

Lemma (subst.o): If

- (ret-hyp) $\vartheta_0 \sqsubseteq_{m+1} \vartheta_1$, and
- (top-hyp) $u[\vartheta_0] \sqsubseteq_m u[\vartheta_1]$ for all values $u[\]$ with operation holes

then $\zeta[\vartheta_0] \sqsubseteq_m \zeta[\vartheta_1]$ for all states $\zeta[\]$ with operation holes.

Proof (subst.o): Again, the proof is analogous to the proof of (subst.c) with the only difference being the hole case. Here $\zeta[\]$ has the form $\gamma \circ \text{appc}([\]) \triangleq u[\]$.

$\gamma[\] \circ \text{appc}(\vartheta_0) \triangle u[\]$ steps uniformly to a smaller computation and by (ret-hyp) and computation

$$\begin{aligned}\gamma[\vartheta_1] \circ \text{appc}(\vartheta_0) \triangle u[\vartheta_1] &\xrightarrow{cm} \gamma[\vartheta_1] \text{appi}(u[\vartheta_1]) \triangle \vartheta_0 \\ &\sqsubseteq_n \gamma[\vartheta_1] \text{appi}(u[\vartheta_1]) \triangle \vartheta_1 \\ &\sqsubseteq_n \gamma[\vartheta_1] \circ \text{appc}(\vartheta_1) \triangle u[\vartheta_1]\end{aligned}$$

□

10.1.4. Extensionality and recursion

Theorem (Extensionality):

$$(\text{ext.op}) \quad \vartheta_0 \sqsubseteq_{n+1} \vartheta_1 \Leftrightarrow (\forall v)(\vartheta_0(v) \sqsubseteq_{n+1} \vartheta_1(v))$$

$$(\text{ext.co}) \quad \gamma_0 \sqsubseteq_{m+1} \gamma_1 \Leftrightarrow (\forall v)(\gamma_0 \triangle v \sqsubseteq_n \gamma_1 \triangle v)$$

Proof (Extensionality): The onlyif directions are instances of the substitution theorem. The if direction follows from **(ext.op)** and **(ext.co)** (see below) by the same argument as for the substitution theorem. □

Lemma (ext.op): If

- (ret-hyp) $\vartheta_0(v) \sqsubseteq_{m+1} \vartheta_1(v)$ for all v , and
- (top-hyp) $u[\vartheta_0] \sqsubseteq_m u[\vartheta_1]$ for all values $u[\]$ with operation holes
then $\zeta[\vartheta_0] \sqsubseteq_m \zeta[\vartheta_1]$ for all states $\zeta[\]$ with operation holes.

Proof (ext.op): As in the proof of **(subst.o)** the only non-automatic case is where $\zeta[\]$ has the form $\gamma[\] \circ \text{appc}([\]) \triangle u[\]$. Here $\gamma[\] \circ \text{appc}(\vartheta_0) \triangle u[\]$ steps uniformly to a smaller computation and by (ret-hyp) and computation

$$\begin{aligned}\gamma[\vartheta_1] \circ \text{appc}(\vartheta_0) \triangle u[\vartheta_1] &\sqsubseteq_m \gamma[\vartheta_1] \triangleright \vartheta_0(u[\vartheta_1]) \\ &\xrightarrow{cm} \gamma[\vartheta_1] \triangleright \vartheta_1(u[\vartheta_1]) \\ &\xrightarrow{cu} \gamma[\vartheta_1] \circ \text{appc}(\vartheta_1) \triangle u[\vartheta_1]\end{aligned}$$

□

Lemma (ext.co): If

- (ret-hyp) $\gamma_0 \triangle v \sqsubseteq_m \gamma_1 \triangle v$ for all v , and
- (top-hyp) $u[\gamma_0] \sqsubseteq_m u[\gamma_1]$ for all values $u[\]$ with continuation holes
then $\zeta[\gamma_0] \sqsubseteq_m \zeta[\gamma_1]$ for all states $\zeta[\]$ with continuation holes.

Proof (ext.co): As in the proof of (**subst.c**) the only non-automatic case is where $\zeta[\]$ has the form $[\] \triangle u[\]$. By the top case and uniform step argument $70 \triangle u[\gamma_0] \text{ Em } 70 \triangle u[\gamma_1]$ and by (ret-hyp) $70 \triangle u[\gamma_1] \sqsubseteq_m \gamma_1 \triangle u[\gamma_1]$. \square

Theorem (Recursion): Let ϑ be a closure of $\lambda f \lambda x. \varphi$ then the recursion operator Rec computes the least fixed-point of ϑ with respect to each refinement of the operational approximation ordering.

$$(\text{fix}) \quad \text{Rec}(\vartheta) \cong_{m+1} \vartheta(\text{Rec}(\vartheta))$$

$$(\text{min}) \quad \vartheta(\vartheta_0) \sqsubseteq_{m+1} \vartheta_0 \Rightarrow \text{Rec}(\vartheta) \sqsubseteq_{m+1} \vartheta_0$$

Proof (Recursion):

Case (Ax): by computation $\text{Rec}(\vartheta)(v) \cong_{m+1} \vartheta(\text{Rec}(\vartheta))(v)$ for any v . Now use extensionality. \square_{fix}

Case (min): By (**rec.min**) (see below) and the argument used in the proof of the substitution theorem we show (by induction on m) that $\zeta[\text{Rec}(\vartheta)] \sqsubseteq_m \zeta[\vartheta_0]$ for any state $\zeta[\]$ with operation holes, and $u[\text{Rec}(\vartheta)] \sqsubseteq_{m+1} u[\vartheta_0]$ for any value $u[\]$ with operation holes. $\square_{\text{min}} \square$

Lemma (rec.min): Assume ϑ is a closure of $\lambda f \lambda x. \varphi$. If

- (ret-hyp) $\vartheta(\vartheta_0) \sqsubseteq_{m+1} \vartheta_0$, and
 - (top-hyp) $u[\text{Rec}(\vartheta)] \sqsubseteq_m u[\vartheta_0]$ for all values $u[\]$ with operation holes
- then $\zeta[\text{Rec}(\vartheta)] \sqsubseteq_m \zeta[\vartheta_0]$ for all states $\zeta[\]$ with operation holes.

Proof (rec.min): As in the proof of (**subst.o**) the only non-automatic case is where $\zeta[\]$ has the form $\gamma[\] \circ \text{appc}([\]) \triangle u[\]$. Assume that $\vartheta = \langle \lambda f. \lambda x. \varphi : \xi \rangle$. Then

$$\begin{aligned} & \gamma[\text{Rec}(\vartheta)] \circ \text{appc}(\text{Rec}(\vartheta)) \triangle u[\text{Rec}(\vartheta)] \\ & \sqsubseteq_m \gamma[\text{Rec}(\vartheta)] \triangleright \langle \varphi : \xi \{ f \leftarrow \text{Rec}(\vartheta), x \leftarrow u[\text{Rec}(\vartheta)] \} \rangle \quad ;; \text{ computation} \\ & \sqsubseteq_m \gamma[\vartheta_0] \triangleright \langle \varphi : \xi \{ f \leftarrow \vartheta_0, x \leftarrow u[\vartheta_0] \} \rangle \quad ;; \text{ induction hypothesis} \\ & \cong_m \gamma[\vartheta_0] \circ \text{appc}(\vartheta(\vartheta_0)) \triangle u[\vartheta_0] \quad ;; \text{ computation} \\ & \text{Em } \gamma[\vartheta_0] \circ \text{appc}(\vartheta_0) \triangle u[\vartheta_0] \quad ;; (\text{ret-hyp}) \end{aligned}$$

cl

Notes

We have defined a sequence of refinements of operational approximation \sqsubseteq_n . The motivation for developing these refinements was that while we have $\chi_0 \sqsubseteq_1 \chi_1$ implies $\chi[\chi_0] \sqsubseteq_1 \chi[\chi_1]$ for χ_0, χ_1 dtrees, values, or continuations and $\chi[\]$ a dtree or value with holes of a suitable sort, it is not the case that under the same assumption $\zeta[\chi_0] \sqsubseteq_1 \zeta[\chi_1]$ for any state $\zeta[\]$ with holes of suitable sort. Thus distinguishable values can be returned by ‘indistinguishable’ dtrees. This holds for each refinement \sqsubseteq_{n+1} . However, by taking the intersection of this chain of approximations we have an approximation relation \sqsubseteq_ω such that for χ_0, χ_1 are dtrees, values, or continuations $\chi_0 \sqsubseteq_\omega \chi_1$ implies $\chi[\chi_0] \sqsubseteq_\omega \chi[\chi_1]$ for $\chi[\]$ any dtree, value or state with holes of a suitable sort.

It is not clear what the importance of this refinement is for our language, since all of the properties we need for our theory hold for each \sqsubseteq_{n+1} , i.e. \sqsubseteq_α provides a model for the theory for any α with $1 \leq \alpha \leq w$. However the ability to refine may become important in the case of languages which have composable continuations or the ability to delimit the scope of continuation capture and aborting (cf. [Felleisen 1988a], [Danvy 1989]).

10.2. Proofs of some simple derived laws

Here we fill in the details for proof of the “simple derived laws” given in the §5.

Theorem (Laws about functions):

- (k-nabs) $(\forall x \in \mathbf{V}^*)(\varphi_0 \cong \varphi_1) \Rightarrow \lambda x.\varphi_0 \cong \lambda x.\varphi_1$
- (op.eta) $f \in \Theta \Rightarrow \lambda z.f(z) \cong f$
- (cmps.id) $f \in \Theta \Rightarrow \text{id} \circ f \cong f \cong f \circ \text{id}$
- (cmps.assoc) $(f \circ g) \circ h \cong f \circ (g \circ h)$
- (bot) $g \in \Theta \Rightarrow \text{Rec}(\lambda f.\lambda x.f(x)) \sqsubseteq g$

Proof (lam.abs): Assume $(\forall x \in \mathbf{V}^*)(\varphi_0 \cong \varphi_1)$ then by **(letv)**

$$(\lambda x.\varphi_0)(x) \cong \mathbf{Q}_0 \cong \varphi_1 \cong (\lambda x.\varphi_1)(x)$$

for all x and by **(op.ext)** we are done. \square

Proof (op.eta): Assume $f \in \Theta$ then by **(letv)** $(\lambda z.f(z))(z) \cong f(z)$ and by **(op.ext)** we are done. \square

Proof (cmps.id,cmps.assoc): by computation and **(op.ext)**. •I

Proof (bot): by **(rec.min)** since by **(op.eta)** $(\lambda f. \lambda x. f(x))(g) \cong g$. \square

Theorem (If laws):

- (if.sort) $\varphi_0 \in U_0 \wedge \varphi_1 \in U_1 \Rightarrow \text{if}(z, \varphi_0, \varphi_1) \in U_0 \cup U_1$
- (if.elim) $\text{if}(z, \varphi, \varphi) \cong Q$
- (if.perm) $\text{if}(x, \text{if}(y, \varphi_a, \varphi_b), \text{if}(y, \varphi_c, \varphi_d)) \cong \text{if}(y, \text{if}(x, \varphi_a, \varphi_c), \text{if}(x, \varphi_b, \varphi_d))$
- (if.lam) $\lambda x. \text{if}(z, \varphi_1, \varphi_2) \cong \text{if}(z, \lambda x. \varphi_1, \lambda x. \varphi_2)$
- (if.subst) $(\varphi \in V^+ \Rightarrow \varphi_1 \cong \varphi_3) \wedge (Q \cong \text{mt} \Rightarrow \varphi_2 \cong \varphi_4) \wedge Q \in V^*$
 $\Rightarrow \text{if}(\varphi, \varphi_1, \varphi_2) \cong \text{if}(\varphi, \varphi_3, \varphi_4)$

Proof (if.sort): by vcases. Assume $\varphi_0 \in U_0$ and $\varphi_1 \in U_1$ then

Case ($z \cong \text{mt}$): by **(ifmt)** and **inlaws** $\text{if}(z, \varphi_0, \varphi_1) \cong \varphi_0 \in U_0 \bigcup U_1$.

Case ($z \in V_+$): by **(ifnmt)** and **inlaws** $\text{if}(z, \varphi_0, \varphi_1) \cong \varphi_1 \in U_0 \bigcup U_1$. \square

Proof (if.elim): by vcases

Case ($z \cong \text{mt}$): $\text{if}(z, \varphi, \varphi) \cong Q$ by **(ifmt)**

Case ($z \in V_+$): $\text{if}(z, Q, Q) \cong Q$ by **(ifnmt)** \square

Proof (if.perm): by vcases using if laws. \square

Proof (if.lam): by vcases using **(op.ext)**. •I

Proof (if.subst): Assume $Q \in V^+ \Rightarrow \varphi_1 \cong \varphi_3$, $Q \cong \text{mt} \Rightarrow \varphi_2 \cong \varphi_4$ and $Q \in V^*$. Then

$$Q \in V^+ \Rightarrow \text{if}(\varphi, \varphi_1, \varphi_2) \cong \varphi_1 \cong Q \cong \text{if}(\varphi, \varphi_3, \varphi_4)$$

and

$$Q \cong \text{mt} \Rightarrow \text{if}(\varphi, \varphi_1, \varphi_2) \cong \varphi_2 \cong \varphi_4 \text{if}(\varphi, \varphi_3, \varphi_4)$$

and by vcases we are done. [Note that the hypothesis $Q \in V^*$ was needed in order for vcases to apply.] \square

Theorem (Cart laws):

$$(\text{cart.assoc}) [[\varphi_0, \varphi_1], \varphi_2] \cong [\varphi_0, [\varphi_1, \varphi_2]]$$

$$(\text{fst.rst}) x \in V^+ \Rightarrow \text{fst}[x, y] \cong \text{fst}(x) \wedge \text{rst}[x, y] \cong [\text{rst}(x), y]$$

Proof (cart.assoc): Choose fresh c, x, y, z with $c \in \mathbb{C}_o$ then by the cart computation laws

$$\begin{aligned}
c([(\varphi_0, \varphi_1), \varphi_2]) &\cong (c \circ \text{carti}(\varphi_2) \circ \text{carti}(\varphi_1))(\varphi_0) \\
c([\varphi_0, [\varphi_1, \varphi_2]]) &\cong (c \circ \text{carti}([\varphi_1, \varphi_2]))(\varphi_0) \\
(c \bullet \text{carti}(\varphi_2) \bullet \text{carti}(\varphi_1))(x) &\cong (c \circ \text{carti}(\varphi_2) \circ \text{cartc}(x))(\varphi_1) \\
(c \circ \text{carti}([\varphi_1, \varphi_2]))(x) &\cong (c \bullet \text{cartc}(x) \bullet \text{carti}(\varphi_2))(\varphi_1) \\
(c \bullet \text{carti}(\varphi_2) \bullet \text{cartc}(x))(y) &\cong (c \circ \text{cartc}([x, y]))(\varphi_2) \\
(c \bullet \text{cartc}(x) \bullet \text{carti}(\varphi_2))(y) &\cong (c \circ \text{cartc}(x) \circ \text{cartc}(y))(\varphi_2) \\
(c \circ \text{cartc}([x, y]))(z) &\cong c[[x, y], z] \\
&\text{and} \\
(c \bullet \text{cartc}(x) \bullet \text{cartc}(y))(z) &\cong c[x, [y, z]]
\end{aligned}$$

hence by **(op.ext)**, **(c.ext)** and the basic cart laws we are done. \square

Proof (fst.rst): Assume $x \in V^+$ then

$$\begin{aligned}
\text{fst}(x) &\in v \quad ;; \text{ in laws} \\
\text{fst}[x, y] &\cong \text{fst}[\text{fst}(x), [\text{rst}(x), y]] \cong \text{fst}(x) \quad ;; \text{ cart and in laws} \\
\text{rst}[x, y] &\cong \text{rst}[\text{fst}(x), [\text{rst}(x), y]] \cong [\text{rst}(x), y] \quad ;; \text{ cart and in laws}
\end{aligned}$$

cl

Theorem (Note laws):

$$\begin{aligned}
(\text{note.triv}) \quad \text{note}(c)\varphi &\cong \varphi \quad ;; c \text{ not free in } \varphi \\
(\text{note.id}) \quad \text{note}(c)c(\varphi) &\cong \varphi \quad ;; c \text{ not free in } \varphi \\
(\text{note.esc}) \quad c \in \mathbb{C}_o \Rightarrow c \circ (\lambda x.\text{note}(c)\varphi) &\cong c \circ (\lambda x.\varphi) \\
(\text{note.ren}) \quad \text{note}(c)\text{note}(c')\varphi &\cong \text{note}(c)\varphi\{c'/c\}
\end{aligned}$$

Proof (note.triv): by **(c.ext)** since by computation $c(\text{note}(c)\varphi) \cong c(\varphi)$. $\bullet \top$

Proof (note.id): by **(c.ext)** since by computation $c(\text{note}(c)c(\varphi)) \cong c(c(\varphi))$ and by (sw') $c(c(\varphi)) \cong c(\varphi)$. \square

Proof (note.esc): by **(op.ext)** since by computation we have

$$(c \bullet \lambda x.\text{note}(c)\varphi)(x) \cong c(\text{note}(c)\varphi) \cong c(\varphi) \cong (c \circ \lambda x.\varphi)(x)$$

cl

Proof (note.ren): by **(c.ext)** since by computation we have

$$c(\text{note}(c)\text{note}(c')\varphi) \cong c(\text{note}(c')\varphi) \cong c(\varphi\{c'/c\}) \cong c(\text{note}(c)\varphi\{c'/c\})$$

□

10.3. Proof of context motion theorem

Theorem (context motion): Let C be an evaluated position context and let c range over continuations. Assume $\text{Frees}[\varphi, x, c, k] \cap \text{trap}(C) = \emptyset$ and x, k are not free in C . Then

$$\begin{aligned} (\text{letx}) \quad & \text{let}\{x \leftarrow \varphi\}C[x] \cong C[\varphi] \\ (\text{escape}) \quad & C[c(\varphi)] \cong c(\varphi) \\ (\text{let.dist}) \quad & C[\text{let}\{x \leftarrow \varphi\}\varphi_0] \cong \text{let}\{x \leftarrow \varphi\}C[\varphi_0] \\ (\text{if.dist}) \quad & C[\text{if}(\varphi, \varphi_1, \varphi_2)] \cong \text{if}(\varphi, C[\varphi_1], C[\varphi_2]) \\ (\text{note.dist}) \quad & C[\text{note}(k)\varphi_0] \cong \text{note}(k)\text{let}\{k \leftarrow k \circ \lambda x. C[x]\}C[\varphi_0] \end{aligned}$$

We begin with proofs of lemmas that correspond to special cases and required for the base case in the induction on context structure.

Lemma (sw'): $c, c' \in \mathbb{C}_o \Rightarrow c(c'(\varphi)) \cong \mathbf{C}'(\mathbf{Q})$

Proof (sw'): Assume $c, c' \in \mathbb{C}_o$ then $c(c'(\varphi)) \cong \text{co appc}(c')(\varphi)$ by (app,appi); $c \circ \text{appc}(c')(x) \cong c(c'(x)) \cong c'(x)$ by (app,appi) and (sw); and $c \circ \text{appc}(c') \cong c'$ by **(op.ext)**. □_{sw'},

Lemma (esc.arg): $c \in \mathbb{C}_o \Rightarrow f(c(\varphi)) \cong \mathbf{C}(\mathbf{Q})$

Proof (esc.arg):

$$\begin{aligned} c'(f(c(\varphi))) & \cong c' \circ \text{appc}(f)(c(\varphi)) \quad \text{;; computation} \\ & \cong c(\varphi) \quad \text{;; (sw')} \\ & \cong c'(c(\varphi)) \quad \text{;; (sw')} \end{aligned}$$

hence by **(c.abs)** we are done. □_{esc.arg}

Lemma (if.arg): $f(\text{if}(\varphi_0, \varphi_1, \varphi_2)) \cong \text{if}(\varphi_0, f(\varphi_1), f(\varphi_2))$

Proof (if.arg): By computation

$$\begin{aligned}
 c(f(\text{if}(\varphi_0, \varphi_1, \varphi_2))) &\cong c \circ \text{appc}(f) \circ \text{ifi}(\varphi_1, \varphi_2)(\varphi_0) \\
 c(\text{if}(\varphi_0, f(\varphi_1), f(\varphi_2))) &\cong c \circ \text{ifi}(f(\varphi_1), f(\varphi_2))(\varphi_0) \\
 x \cong \text{mt} \Rightarrow c \circ \text{appc}(f) \circ \text{ifi}(\varphi_1, \varphi_2)(x) &c \circ \text{appc}(f)(\varphi_2) \cong c \circ \text{ifi}(f(\varphi_1), f(\varphi_2))(x) \\
 x \in V^+ \Rightarrow c \circ \text{appc}(f) \circ \text{ifi}(\varphi_1, \varphi_2)(x) &c \circ \text{appc}(f)(\varphi_1) \cong c \circ \text{ifi}(f(\varphi_1), f(\varphi_2))(x)
 \end{aligned}$$

Thus by value cases

$$c \circ \text{appc}(f) \circ \text{ifi}(\varphi_1, \varphi_2)(x) \cong c \circ \text{ifi}(f(\varphi_1), f(\varphi_2))(x)$$

and by **(c.ext)** and **(op.ext)** (taking c, x fresh) we are done. $\square_{\text{if.arg}}$

Lemma (note.if): If c is not free φ_0 then

$$\text{note}(c)\text{if}(\varphi_0, \varphi_1, \varphi_2) \cong \text{if}(\varphi_0, \text{note}(c)\varphi_1, \text{note}(c)\varphi_2)$$

Proof (note.if): By computation we have

- (i) $c(\text{note}(c)\text{if}(\varphi_0, \varphi_1, \varphi_2)) \cong c \circ \text{ifi}(\varphi_1, \varphi_2)(\varphi_0)$
- (ii) $c(\text{if}(\varphi_0, \text{note}(c)\varphi_1, \text{note}(c)\varphi_2)) \cong c \circ \text{ifi}(\text{note}(c)\varphi_1, \text{note}(c)\varphi_2)(\varphi_0)$
- (iii) $x \cong \text{mt} \Rightarrow c \circ \text{ifi}(\varphi_1, \varphi_2)(x) \cong c(\varphi_2) \cong c \circ \text{ifi}(\text{note}(c)\varphi_1, \text{note}(c)\varphi_2)(x)$
- (iv) $x \in V^+ \Rightarrow c \circ \text{ifi}(\varphi_1, \varphi_2)(x) \cong c(\varphi_1) \cong c \circ \text{ifi}(\text{note}(c)\varphi_1, \text{note}(c)\varphi_2)(x)$

Thus by value cases we have

$$c \circ \text{ifi}(\varphi_1, \varphi_2)(x) \cong c \circ \text{ifi}(\text{note}(c)\varphi_1, \text{note}(c)\varphi_2)(x)$$

and by **(c.abs)** (using c not free in φ_0) and **(op.ext)** we are done. $\square_{\text{note.if}}$

Lemma (note.arg): $f \in \mathbb{O} \Rightarrow f(\text{note}(c)\varphi) \cong \text{note}(c)\text{let}\{c \leftarrow c \circ f\}f(\varphi)$

Proof (note.arg): Assume $f \in \mathbb{O}$ then by computation

$$\begin{aligned}
 c(f(\text{note}(c)\varphi)) &\cong (c \circ \text{appc}(f))(\text{note}(c)\varphi) \\
 &\cong (c \circ f)(\text{note}(c)\varphi) \quad ;; \text{ using } \mathbf{(op.eta)} \lambda x. f(x) \cong f \\
 &\cong (c \circ f)(\text{let}\{c \leftarrow c \circ f\}\varphi) \\
 &\cong c(f(\text{let}\{c \leftarrow c \circ f\}\varphi)) \\
 &\cong c(\text{let}\{c \leftarrow c \circ f\}f(\varphi)) \quad ;; \text{ two applications of } (\mathbf{letv}) \\
 &\cong c(\text{note}(c)\text{let}\{c \leftarrow c \circ f\}f(\varphi))
 \end{aligned}$$

and by $c.\text{abswearedone} \vdash \square \diamond \mathbb{M} \oplus \mathbb{G} \square \mathbb{N}$

Corollary (note.let): Taking $f = \lambda x.\varphi_0$ with c not free in φ_0 we have

$$\text{let}\{x \leftarrow \text{note}(c)\varphi\} \varphi_0 \cong \text{note}(c) \text{let}\{c \leftarrow c \circ \lambda x.\varphi_0\} \text{let}\{x \leftarrow \varphi\} \varphi_0$$

Proof (letx): Assume $\text{Frees}[x, \varphi_0] \cap \text{trap}(C) = \emptyset$ and x is not **free** in C . We prove

$$\text{let}\{x \leftarrow \varphi_0\} C[x] \cong C[\varphi_0]$$

by induction on construction of **C**. We consider a few sample cases.

Case ([]): $\text{let}\{x \leftarrow \varphi_0\} x \cong \varphi_0$ by **(id)**. \square

Case ($C_0(\varphi_1)$): Choose fresh $c \in \mathbb{C}_0$ then by computation and induction hypothesis

$$\begin{aligned} c(\text{let}\{x \leftarrow \varphi_0\} C_0[x](\varphi_1)) &\cong (c \circ \text{appc}(\lambda x.C_0[x](\varphi_1)))(\varphi_0) \\ c(C_0[\varphi_0](\varphi_1)) &\cong (c \circ \text{appi}(\varphi_1))(C_0[\varphi_0]) \\ &\cong (c \circ \text{appi}(\varphi_1))((\lambda x.C_0[x])(\varphi_0)) \\ &\cong (c \circ \text{appi}(\varphi_1) \circ \text{appc}(\lambda x.C_0[x]))(\varphi_0) \\ c \circ \text{appi}(\varphi_1) \circ \text{appc}(\lambda x.C_0[x])(x) &\cong c \circ \text{appi}(\varphi_1)C_0[x] \\ &\cong c \circ \text{appc}(\lambda x.C_0[x](\varphi_1))(x) \quad ;; \text{ using } 2 \text{ not free } C[] \end{aligned}$$

and by **(c.ext, o.ext)** we are done. \square

Case ($\text{if}(v, C_1, C_2)$): by **vcases** Assume $v \cong \text{mt}$ then by computation and induction hypothesis

$$\begin{aligned} \text{let}\{x \leftarrow \varphi_0\} \text{if}(v, C_1[x], C_2[x]) &\cong \text{let}\{x \leftarrow \varphi_0\} C_2[x] \\ &\cong C_2[\varphi_0] \\ &\cong \text{if}(v, C_1[\varphi_0], C_2[\varphi_0]) \end{aligned}$$

Similarly we have

$$v \in V^+ \Rightarrow \text{let}\{x \leftarrow \varphi_0\} \text{if}(v, C_1[x], C_2[x]) \cong \text{if}(v, C_1[\varphi_0], C_2[\varphi_0])$$

\square

Case ($\text{let}\{z \leftarrow v\} C_0$): Let $C_1 = C_0\{z/v\}$ then by computation and induction hypothesis

$$\text{let}\{x \leftarrow \varphi_0\} \text{let}\{z \leftarrow v\} C_0[x] \cong \text{let}\{x \leftarrow \varphi_0\} C_1[x] \cong C_1[\varphi_0] \cong \text{let}\{z \leftarrow v\} C_0[\varphi_0]$$

where the freeness condition is used in the first and third steps (let conversions).

□

Case (note(c)&): by computation and induction hypothesis

$$\begin{aligned} c(\text{let}\{x \leftarrow \varphi_0\} \text{note}(c)C_0[x]) &\cong (c \circ \text{appc}(\lambda x. \text{note}(c)C_0[x]))(\varphi_0) \\ c(\text{note}(c)C_0[\varphi_0]) &\cong c(C_0[\varphi_0]) \cong c(\text{let}\{x \leftarrow \varphi_0\}C_0[x]) \\ &\cong (c \circ \text{appc}(\lambda x. C_0[x]))(\varphi_0) \\ c \circ \text{appc}(\lambda x. \text{note}(c)C_0[x])(x) &\cong c(\text{note}(c)C_0[x]) \\ &\cong c(C_0[x]) \cong c \circ \text{appc}(\lambda x. C_0[x])(x) \end{aligned}$$

hence by (c.ext, o.ext) we are done. □

The cases $f(C_1)$, $[C_0, \varphi_1]$, $[v, C_1]$, $\text{if}(C_0, \varphi_1, \varphi_2)$, $\text{fst}(C_0)$, $\text{rst}(C_0)$, are similar to the case $C_0(\varphi_1)$.

□**letx**

Proof (escx): Let c range over continuations and assume $\text{Frees}[c, \varphi_0] \cap \text{trap}(C) = \emptyset$. Choose x fresh then

$$C[c(\varphi_0)] \cong \text{let}\{x \leftarrow c(\varphi_0)\}C[x] \cong c(\varphi_0)$$

by **(letx)** and **(esc.arg)**. □**escx**

Proof (let.dist): Assume $\text{Frees}[x, \varphi_0] \cap \text{trap}(C) = \emptyset$ and x is not free in C . Then

$$\begin{aligned} C[\text{let}\{x \leftarrow \varphi_0\}\varphi] &= C[(\lambda x. \varphi)(\varphi_0)] \\ &\cong \text{let}\{x \leftarrow \varphi_0\}C[(\lambda x. \varphi)(x)] \quad \text{;; (letx)} \\ &\cong \text{let}\{x \leftarrow \varphi_0\}C[\varphi] \quad \text{;; (letv)} \end{aligned}$$

□ **let.dist**

Proof (if.dist): Assume $\text{Frees}(\varphi_0) \cap \text{trap}(C) = \emptyset$. We will prove

$$C[\text{if}(\varphi_0, \varphi_1, \varphi_2)] \cong \text{if}(\varphi_0, C[\varphi_1], C[\varphi_2])$$

by induction on the number of trapped variables in C . First we note that (by induction on construction of C) either C has no trapped variables (is a pure evaluation context) or we can find C_0, C_1 such that $\text{trap}(C_0) = \emptyset$ and such that $C = C_0[\text{let}\{x \leftarrow v\}C_1]$ or $C = C_0[\text{note}(c)C_1]$

Case (pure): Choose x fresh then

$$\begin{aligned} C[\text{if}(\varphi_0, \varphi_1, \varphi_2)] &\cong \text{let}\{x \leftarrow \text{if}(\varphi_0, \varphi_1, \varphi_2)\}C[x] &&; \text{ letx} \\ &\cong \text{if}(\varphi_0, \text{let}\{x \leftarrow \varphi_1\}C[x], \text{let}\{x \leftarrow \varphi_2\}C[x]) &&; \text{ if.arg} \\ &\cong \text{if}(\varphi_0, C[\varphi_1], C[\varphi_2]) &&; \text{ letx} \end{aligned}$$

Case (note):

$$\begin{aligned} C[\text{if}(\varphi_0, \varphi_1, \varphi_2)] &= C_0[\text{note}(c)C_1[\text{if}(\varphi_0, \varphi_1, \varphi_2)]] \\ &\cong C_0[\text{note}(c)\text{if}(\varphi_0, C_1[\varphi_1], C_1[\varphi_2])] &&; \text{ induction hypothesis} \\ &\cong C_0[\text{if}(\varphi_0, \text{note}(c)C_1[\varphi_1], \text{note}(c)C_1[\varphi_2])] &&; (\text{note.if}) \\ &\cong \text{if}(\varphi_0, C[\varphi_1], C[\varphi_2]) &&; \text{ pure case} \end{aligned}$$

□

Case (let):

$$\begin{aligned} C[\text{if}(\varphi_0, \varphi_1, \varphi_2)] &= C_0[\text{let}\{x \leftarrow v\}C_1[\text{if}(\varphi_0, \varphi_1, \varphi_2)]] \\ &\cong C_0[\text{let}\{x \leftarrow v\}\text{if}(\varphi_0, C_1[\varphi_1], C_1[\varphi_2])] &&; \text{ induction hypothesis} \\ &\cong C_0[\text{if}(\varphi_0, \text{let}\{x \leftarrow v\}C_1[\varphi_1], \text{let}\{x \leftarrow v\}C_1[\varphi_2])] \\ &\quad ;; (\text{letv}) \text{ twice using } 2 \text{ not free } \varphi_0 \\ &\cong \text{if}(\varphi_0, C[\varphi_1], C[\varphi_2]) &&; \text{ pure case} \end{aligned}$$

□

□**if.dist**

Proof (note.dist): Assume c is not in $\text{Frees } C \cup \text{trap}(C)$. We prove

$$C[\text{note}(c)\varphi] \cong \text{note}(c)\text{let}\{c \leftarrow c \circ \lambda x.C[x]\}C[\varphi]$$

by induction on number of trapped variables

Case (pure):

$$\begin{aligned} C[\text{note}(c)\varphi] &\\ &\cong \text{let}\{x \leftarrow \text{note}(c)\varphi\}C[x] &&; (\text{letx}) \text{ choosing } x \text{ fresh} \\ &\cong \text{note}(c)\text{let}\{c \leftarrow c \circ \lambda x.C[x]\}(C[\varphi]) &&; (\text{note.let}), (\text{letx}) \end{aligned}$$

□pure

Case (note):

$$\begin{aligned}
 C[\text{note}(c)\varphi] &= C_0[\text{note}(k)C_1[\text{note}(c)\varphi]] \quad ;\text{ since } C_0 \text{ is pure} \\
 &\cong C_0[\text{note}(k)\text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C_1[x]\}C_1[\varphi]] \quad ;\text{ induction} \\
 &\cong C_0[\text{note}(c)\text{note}(k)\text{let}\{c \dashv k \circ \lambda x.C_1[x]\}C_1[\varphi]] \quad ;\text{ note.ren} \\
 &\cong C_0[\text{note}(c)\text{note}(k)\text{let}\{c \dashv k \circ \lambda x.\text{note}(k)C_1[x]\}C_1[\varphi]] \quad ;\text{ note.escape} \\
 &\cong C_0[\text{note}(c)\text{note}(k)\text{let}\{c \dashv c \circ \lambda x.\text{note}(k)C_1[x]\}C_1[\varphi]] \quad ;\text{ note.ren} \\
 &\cong C_0[\text{note}(c)\text{let}\{c \dashv c \circ \lambda x.\text{note}(k)C_1[x]\}\text{note}(k)C_1[\varphi]] \quad ;\text{ (letv) twice} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C_0[x]\} \\
 &\quad C_0[\text{let}\{c \dashv c \circ \lambda x.\text{note}(k)C_1[x]\}\text{note}(k)C_1[\varphi]] \\
 &\quad ;\text{ pure case} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C_0[x] \circ \lambda x.\text{note}(k)C_1[x]\}C_0[\text{note}(k)C_1[\varphi]] \\
 &\quad ;\text{ (letv)} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C[x]\}C[\varphi] \quad ;\text{ (letx)}
 \end{aligned}$$

□

Case (let):

$$\begin{aligned}
 C[\text{note}(c)\varphi] &= C_0[\text{let}\{y \dashv v\}C_1[\text{note}(c)\varphi]] \\
 &\cong C_0[\text{let}\{y \dashv v\}\text{note}(c)\text{let}\{c \dashv c \circ C_1\}C_1[\varphi]] \quad ;\text{ induction hypothesis} \\
 &\cong C_0[\text{note}(c)\text{let}\{c \dashv c \circ \lambda x.\text{let}\{y \dashv v\}C_1[x]\}\text{let}\{y \dashv v\}C_1[\varphi]] \\
 &\quad ;\text{ (letv)} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C_0[x]\} \\
 &\quad C_0[\text{let}\{c \dashv c \circ \lambda x.\text{let}\{y \dashv v\}C_1[x]\}\text{let}\{y \dashv v\}C_1[\varphi]] \\
 &\quad ;\text{ pure case} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C_0[x] \circ \lambda x.\text{let}\{y \dashv v\}C_1[x]\}C_0[\text{let}\{y \dashv v\}C_1[\varphi]] \\
 &\quad ;\text{ (let .dist)} \\
 &\cong \text{note}(c)\text{let}\{c \dashv c \circ \lambda x.C[x]\}C[\varphi] \quad ;\text{ (letx)}
 \end{aligned}$$

□

□note.dist

10.4. Representation of Computation

We **define** a syntactic representation S of dtrees, values, continuations, and states and show that computations can be simulated by chains of operational equivalences using the computation and data laws. Let ξ_D be the environment mapping constants to the corresponding **data and data operations**. We assume that each data element **d is the value of a closed data form** and let $S(d)$ be some such form. Then S is extended to the remaining semantic **entities as follows**

$$\begin{aligned} S([a_1, \dots, a_n]) &= [S(a_1), \dots, S(a_n)] \\ S(\xi)(x) &= S(\xi(x)) \\ S(\langle \varphi : \xi \rangle) &= S(\xi)(\varphi) \\ S(\text{top}) &= \text{top} \\ S(\gamma \circ (\text{appi}(\varphi) : \xi)) S(\gamma) \circ \text{appi}(S(\langle \varphi : \xi \rangle)) \\ &\dots \\ S(\gamma \triangleright \delta) &= \text{app}(S(\gamma), S(\delta)) \\ S(\gamma \triangleleft v) &= \text{app}(S(\gamma), S(v)) \end{aligned}$$

where $S(\xi)(\varphi)$ denotes the natural extension of a map **from** variables to forms to a map from forms to forms — i.e. substitution for free occurrences of variable symbols.

Theorem (synrep):

m(i) $\langle S(\chi) : \xi_D \rangle \cong \chi$ for χ a dtree, continuation or value.

- (ii) If $\zeta_0 \rightarrowtail \zeta_1$ then $S(\zeta_0) \cong S(\zeta_1)$ is provable from the computation and data laws.

Proof (synrep.i): By induction on the structure of dtrees, continuations, and values.

Case (data): $(S(d) : \xi_D) \cong d$ by definition of $S(d)$.

Case (cart):

$$\langle S([a_1, \dots, a_n]) : \xi_D \rangle = \langle [S(a_1), \dots, S(a_n)] : \xi_D \rangle \cong [a_1, \dots, a_n]$$

since by the induction hypothesis $\langle S(a_i) : \xi_D \rangle \cong a_i$.

Case (dtree,pfn): If $\chi = \langle \varphi : \xi \rangle$ and x_1, \dots, x_n are the variables in the domain of ξ then

$$\begin{aligned} \langle S(\chi) : \xi_D \rangle &= \langle S(\xi)(\varphi) : \xi_D \rangle \\ &\cong \langle \text{let}\{x_1 \ 4 \ S(\xi(x_1))\} \dots \text{let}\{x_n \ n + S(\xi(x_n))\} \varphi : \xi_D \rangle \quad \text{;; by (letv)} \\ &\cong \langle \varphi : \xi \rangle \quad \text{;; by induction hypothesis and computation} \end{aligned}$$

Case (continuation): If $\gamma = \gamma_0 \circ \langle \text{appi}(\varphi) : \xi \rangle$ then

$$\langle S(\gamma) : \xi_D \rangle = \langle S(\gamma_0) \circ S(\xi)(\text{appi}(\varphi)) : \xi_D \rangle \cong 7$$

since by induction hypothesis $\langle S(\gamma_0) : \xi_D \rangle \cong \gamma_0$ and $\langle S(\xi)(\text{appi}(\varphi)) : \xi_D \rangle \cong \langle \text{appi}(\varphi) : \xi \rangle$. \square_i

Proof (synrep.ii): It suffices to consider single steps $\zeta_0 \rightarrow_i \zeta_1$ and cases according to the rule applied (as given in Figure 5). In the cases **sym**, **lam**, **mt**, and **top** we have $S(\zeta_0)$ and $S(\zeta_1)$ are identical. The **cases** **app**, **appi**, **if**, **cart**, **carti**, **fst**, and **rst** follow directly from the corresponding computation laws. For the cases **o**, **appc**, and **sw** we use the **app**, and **appi** laws in reverse to obtain

$$(S(7) \circ \text{appc}(S(\vartheta)))(S(v)) \cong S(\gamma)(S(\vartheta)(S(v)))$$

Then we use the data laws, the **letv** law, or the **sw** law according to the sort of ϑ . For the **cartc**, **fstc**, and **rstc** cases we run the computation rules backwards as for the **appc** case and then use the sequence rules. For the **ifi** case we use the **ifi** laws and the fact that any closed value form can be put in a ‘canonical form’ using the sequence rules, thus emptiness is decidable for such forms within the theory. Finally the **note** case follows from the **note** law. \square_{ii}

10.5. Relation to standard definition of operational relations

The standard definition of operational equivalence is trivial equivalence in all closing contexts. More precisely, for fixed \mathfrak{D} , we define standard operational approximation \sqsubseteq^* by $\varphi_0 \sqsubseteq^* \varphi_1$ just if

$$\text{top} \triangleright \langle C[\varphi_0] : \xi_D \rangle \sqsubseteq_0 \text{top} \triangleright \langle C[\varphi_1] : \xi_D \rangle$$

for all contexts C (forms with holes in arbitrary positions) such that $C[\varphi_0], C[\varphi_1]$ are closed (contain only data or data operation constants as free symbols).

Theorem (standard): $\varphi_0 \sqsubseteq \varphi_1 \Leftrightarrow \varphi_0 \sqsubseteq^* \varphi_1$

Proof (standard): The onlyif direction follows from the fact that \sqsubseteq is a congruence on forms. For the if direction we use the fact that for any continuation γ and environment ξ (extending ξ_D) there is a context C such that

$$\text{top} \triangleright C[\varphi] : \xi_D \rightarrow 7 \triangleright (\varphi : \xi)$$

for any form φ closed by ξ . \square