August 1985

Report No. STAN-CS-85-1060

The Essence of Rum
A Theory of the intensional and extensional aspects of Lisp-type computation

by

Carolyn L. Talcott

Department of Computer Science

Stanford University
Stanford, CA 94305

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Essence of Rum A Theory of the intensional and extensional aspects of Lisp-type computation		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER STAN-CS-85-1060
7. AUTHOR(s) Carolyn L. Talcott		8. CONTRACT OR GRANT NUMBER(s) N00039-82-C-0250
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University, Stanford CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency ,		12. REPORT DATE August 1985 13. NUMBER OF PAGES 249
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Lisp-type	closure
intensional	continuation
extensional	computation-structure
program transformation	comparison relation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(on reverse side)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Rum is a theory of applicative, side-effect free computations over an algebraic data structure. It goes beyond a theory of functions computed by programs, treating both intensional and extensional aspects of computation. Powerful programming tools such as streams, object-oriented programming, escape mechanisms, and co-routines can be represented. Intensional properties include the number of multiplications executed, the number of context switches, and the maximum stack depth required in a computation. Extensional properties include notions of equality for streams and co-routines and characterization of functionals implementing strategies for searching tree-structured spaces. Precise definitions of informal concepts such as stream and co-routine are given and their mathematical theory is developed. Operations on programs treated include program transformations which introduce functional and control abstractions; a compiling morphism that provides a representation of control abstractions as functional abstractions; and operations that transform intensional properties to extensional properties. The goal is not only to account for programming practice in Lisp, but also to improve practice by providing mathematical tools for developing programs and building programming systems.

Rum views computation as a process of generating computation structures – trees for context-independent computations and sequences for context-dependent computations. The recursion theorem gives a fixed-point function that computes computationally minimal fixed points. The context insensitivity theorem says that context-dependent computations are uniformly parameterized by the calling context and that computations in which context dependence is localized can be treated like context-independent computations. Rum machine structure and morphism are introduced to define and prove properties of compilers. The hierarchy of comparison relations on programs ranges from intensional equality to maximum approximation and equivalence relations that are extensional. The fixed-point function computes the least fixed point with respect to the maximum approximation. Comparison relations, combined with the interpretation of programs using computation structures, provide operations on programs both with meanings to preserve and meanings to transform.

The Essence of Rum

A theory of the intensional and extensional aspects of Lisp-type computation

Carolyn Talcott
Stanford University

Research supported by ARPA contract N00039-82-C-0250.

Acknowledgements

I would like to thank Professor Solomon Feferman for agreeing to undertake the task of being my my thesis advisor, and for his interest, encouragement, and support. I am grateful to him for many helpful discussions which have provided a deeper understanding of relations and distinctions between foundational work in logic and computer science. These discussions have been fundamental in clarifying the ideas, motivations, and goals of my work and in relating it to work in logic.

I would like to thank a number of other people who provided ideas and helpful input at various stages in the development of Rum.

Professor John McCarthy, my original thesis advisor, has been a constant source of ideas and has suggested many examples and informal notions that served as starting points for much of my work. His criticism of my presentation of the work in seminars and in writing has been enormously valuable. He has also provided research support throughout the course of this work.

Many of the basic notions of Rum evolved from numerous discussions with Richard Weyhrauch about computation and about how a computation world should be put together. He convinced me of the value of closures and continuations as practical programming tools, and of the importance of separate treatment of computation mechanisms and data structures. He has listened tirelessly to explanations of my ideas as they evolved, and played an essential role in the process of debugging these ideas.

Professor Georg Kreisel, through seminars and conversations, has sharpened my critical facilities and my understanding of subtle but crucial distinctions in foundational work in mathematics. The set of examples used to give an introduction to Rum were inspired by conversations with him about features of Lisp computation.

Shuzo Takahashi provided valuable input by carrying out a formalization of parts of Rum in FOL; Joe Weening provided a practical co-routine example and explained its use and operation; and N. Shankar convinced me of the effectiveness of the deBruijn notation in formalizing properties of abstraction and application. Virginia Kean provided technical writing assistance essential in producing a first draft of the thesis; and Ian Mason and Martin Brooks gave many valuable critical comments on early drafts. I am especially grateful to Michael Beeson and Peter Mosses for careful reading of the final draft and for pointing out mistakes, misleading statements, and other points of confusion.

Finally and most importantly I wish to thank my husband, John McCarthy, for his unflagging patience, encouragement, and confidence in me during the final years of research and writing.

Table of Contents

Table of Contents

List of Figures

Introduction

Rum is a theory of the description and structure of applicative, side-effect free computations over an arbitrary algebraic data structure. This theory goes beyond a theory of functions computed by programs, providing tools for treating both intensional and extensional aspects of computation. Properties of powerful programming tools such as functions as values, streams, object oriented programming, escape mechanisms, and co-routines can be represented. Precise definitions of informal concepts such as stream and co-routine are given and their mathematical theory is developed. The point is not only to account for programming practice, but also to improve practice by providing mathematical tools for developing programs and building programming systems. Among the intensional properties that can be treated are the number of multiplications executed, the number of context switches, and the maximum stack depth required in a computation. Among the extensional properties are notions of equality for streams and co-routines and characterization of functionals implementing strategies for searching tree-structured spaces. A variety of operations on programs are treated including program transformations which introduce functional and control abstractions; a compiling morphism that provides a representation of control abstractions as functional abstractions; and operations that transform intensional properties to extensional properties. There is a rich hierarchy of approximation and equivalence relations on programs ranging from equivalence as descriptions of computation to equivalence as functions. These relations, combined with the interpretation of programs using computation structures, provide operations on programs both with meanings to preserve and meanings to transform.

An important motivation and guide has been the desire to understand the construction and use of computation systems such as Lisp. We call this body of work *symbolic computation.* Our goal is to provide a mathematical context where diverse aspects of computation can be represented and new ideas can be explored and developed. A few such aspects are

- functions as values – the value of a lambda expression in an environment is a closure with the interpretation of free variables of the expression fixed to be that given by the evaluation environment. Functional abstractions such as closures can be used to represent structured data such as tuples and streams, to represent the continuation of a computation, to describe delayed or lazy evaluation, and to represent objects in the object oriented style of computation where objects contain the information about operations that can be applied to them rather than the usual situation where operations contain information about the objects to which they apply.

- computation contexts as values – objects called continuations represent computation contexts – the component of computation states that describes how computations are to continue. Control abstractions such as continuations can be used to describe computation mechanisms such as non-local jumps (escaping) and co-routining.
- programs as data – programs may operate on other programs to interpret, compile, optimize, expand macro definitions, and to generate derived programs that compute intensional properties of a given program
- computation states as data – used in tools for writing and debugging programs such as program editors, tracing, and in tools for interacting with a machine during a computation such as single steppers, breakpoints and the Lisp command *baktrace* which presents a brief summary of the events leading to the current computation state.

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 rum-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.

Our work draws on ideas and results from several areas of computer science and logic. Of particular significance for the present work are (in chronological order)

Kleene - the basic concepts of recursion theories and their formalization

McCarthy - conditional expressions as recursive descriptions of computation

Landin - closures as interpretation of lambda expressions in an environment

Moschovakis - recursion on abstract structures

Scott - extensional models of the lambda calculus

Morris, Wadsworth - lambda abstraction to represent computation contexts

Feferman - non-extensional theories and inductively presented formal systems

In addition many ideas have come from studying examples of computing with closures and continuations provided by Burstall, Sussman, Steele, Friedman, Wand and others.

In order to facilitate natural representation of diverse aspects of computation, $\mathcal{R}um$ has a rich ontology with a mixture of syntactic and semantic notions. Symbolic expressions called *forms* are the basic syntactic entities. A computation is described by a form closed in an *environment* that assigns values to the free symbols of the form. Computation primitives include conditional, application, abstraction and sequence formation. There are functional abstractions called *pfns* which are analogous to partial functions but contain information about *how* they are to be computed. There are control abstractions called *continuations* which represent computation contexts. In order to treat both functional and context dependent aspects of computation naturally, we consider two basic structures for representing computation – trees and sequences. Tree-structured computation is characterized by two relations: *evaluation* which relates descriptions to the value returned by the computation described and *subcomputation* which provides the tree structure of computation. An important subrelation of *subcomputation* is *reduces-to* which identifies subcomputations that can simply replace the main computation, rather than returning a value to a saved context. Sequential computation is carried out by generating sequences of *computation states* using the *step* relation. Computation states are composed of a continuation and a current subcomputation. The structure of computation states and the step relation are derived naturally from tree-structured computation. This has the consequence that many tools for proving properties of tree-structured computation can be generalized to programs in which context dependence is localized.

Two basic results about tree-structured computation are the recursion theorem and a computation induction principle. The recursion theorem gives a recursion pfn that computes a computationally least fixed point of pfns which compute functionals and thus provides a means of definition by recursion. Computation induction expresses the fact that the subcomputation relation is well-founded for computations which return a value and thus provides a tool for proving properties of programs. An important theorem about sequential computation is that the computation described by a form closed in an environment will, uniformly with respect to the context, either return a value to any calling context, transfer control to another context, or not return a value.

A class of comparison relations is defined for tree-structured computation that includes both approximation and equivalence relations on descriptions. Each comparison relation corresponds to forgetting selected details of computation while preserving the evaluation and application structure. Methods are developed for construction of comparison relations and for proving properties of particular comparisons. There is a maximum approximation relation $\sqsubseteq$ and a maximum equivalence relation $\cong$. Two key theorems about these relations are (i) $\sqsubseteq$ and $\cong$ are extensional and (ii) the recursion pfn computes the least fixed point with respect to $\sqsubseteq$. Extensionality of $\sqsubseteq$ means that for any pair of pfns, φ_0 and φ_1, φ_0 approximates φ_1 iff φ_0 applied to any argument v approximates φ_1 applied to v.

$$\varphi_0 \sqsubseteq \varphi_1 \leftrightarrow (\forall v)(\varphi_0{}' v \sqsubseteq \varphi_1{}' v)$$

By a similar result for $\cong$, pfns describe partial functions on $\cong$-equivalence classes of the computation domain and two pfns are $\cong$-equivalent iff they describe the same partial function. There is a rich hierarchy of comparisons between equality and the maximal relations. For example, work preserving transforms such as distribution of conditional over conditional ($\mathfrak{f}_0 \mapsto \mathfrak{f}_1$ reads "$\mathfrak{f}_0$ transforms to $\mathfrak{f}_1$")

$$\mathsf{if(if(p(x),q(x),r(x)),g(x),h(x))} \mapsto \mathsf{if(p(x),if(q(x),f(x),g(x)),if(r(x),f(x),g(x)))}$$

give rise to comparisons in which related forms describe computation trees with the same number of nodes of each sort, but different subcomputation structures. The theory of comparison relations together with the interpretation of descriptions by rules for generating computation structures provides concepts and tools useful in developing a theory of program specifications and transformations.

In order to compare computations carried out by different processes, notions of *Rum* machine structure and morphism are introduced. This serves as a paradigm for defining and proving properties of compilers. A machine structure has states and a step relation, with states naturally generated from a class of symbolic descriptions. A morphism † maps states ς of the source machine $\mathcal{A}$ to states $\varsigma^\dagger$ of the target machine $\mathcal{B}$ in a manner that carries the step relation $\rightarrowtail_{\mathcal{A}}$ of the source machine to steps $\rightarrowtail_{\mathcal{B}}$ of the target machine.

$$\begin{tikzcd}
\varsigma_0 \arrow[r, phantom, "\rightarrowtail_{\mathcal{A}}"] \arrow[d] & \varsigma_1 \arrow[d] \\
\varsigma_0^\dagger \arrow[r, phantom, "\rightarrowtail_{\mathcal{B}}"] & \varsigma_1^\dagger
\end{tikzcd}$$

Sequential computation has a natural machine structure $\mathcal{R}$. A richer machine structure $\mathcal{T}$ based on *sequential descriptions* is defined within the tree-structured computation model. The key characteristic of sequential descriptions is that the *evaluation* relation restricted to sequential descriptions is just the *reduces-to* relation, making *reduces-to* a natural step relation. The main result is the existence of a *Rum* machine morphism mapping $\mathcal{R}$ to $\mathcal{T}$. This morphism corresponds to the normal form theorem of recursion theory and it makes the relation between functional and control abstractions precise.

Working in *Rum*. In addition to developing general tools for reasoning about computation, a variety of examples have been worked out to illustrate the use of these tools and to demonstrate the adequacy of the theory. In particular we have formulated and proved the correctness of the following

- a program that uses continuations as an escape mechanism to avoid unnecessary work in computing the product of the numbers in a tree with numbers at the leaves. If a zero is encountered it is returned directly to the caller rather than passing the information back in the normal fashion to be rechecked at every level. This corresponds to the use of *catch* and *throw* in Lisp.
- a simple co-routine derived from a piece of network software to convert a stream of bits presented as 36-bit words to a stream of 32-bit words. Continuations are used to implement the co-routine mechanism.
- a pattern matcher that generates a stream consisting of all matches of an object to a pattern. The pattern matcher uses pfns to remember the current state of the search and to implement backtracking.
- a derivation map that transforms source programs to derived programs computing properties of the computation described by the source programs.

Plan

The contents of this thesis fall into three major segments. Chapters I-III contain background and introductory material. The main body of the work is presented in Chapters IV-VII. Chapter VIII contains a summary of the work presented and together with Appendices A and B fills in some gaps.

In Chapter I we outline the origins of symbolic computation and the basic goals of a mathematical theory of computation. Examples of programs and operations on programs are given that help introduce some of the informal concepts and programming tools we wish to treat. The main concepts and results from programming language semantics and logic that form the foundation for our work are surveyed. In the final section the goals of our work are presented. Chapter II is an informal introduction to the structures and concepts of *Rum*, illustrated by a series of simple examples. The mathematical tools and notation used in the remaining chapters are summarized in Chapter III. Tree-structured computation is described in Chapter IV. The basic objects, operations, and relations are formally defined and the connection between computation trees and the relations *evaluation* and *subcomputation* characterizing these structures is given. Some additional notation for working in *Rum* is developed. A small library of pfns and their properties is begun including algebraic combinators, a recursion pfn, and schemes for recursion on sequences. The notion of *stream* is defined as a set of pfns and some definitions and properties of operations on streams are given. Sequential computation is described in Chapter V. Additional rules for generating objects are introduced and additional operations and relations are defined, extending the world of tree-structured computation. A theorem expressing the uniform dependence on context of sequential computations is proved and relations and theorems for tree-structured computation are extended to sequential computation. The use of continuations to represent co-routines is explained, basic properties of co-routines are given, and a simple co-routine is defined and proved correct. In Chapter VI the class of comparison relations is defined and studied. A summary of algebraic operations (such as union and intersection) on comparisons and of properties of comparisons preserved by these operations is given. Theorems providing methods for constructing and extending comparisons are proved. These are applied to prove extensionality of the maximum approximation and the maximum equivalence relations, to prove the least fixed point theorem for the recursion pfn, to prove that the computational characterization of recursion uniquely determines the recursion pfn modulo the maximum equivalence, and to generate a variety of comparisons corresponding to typical program transformations. Chapter VII treats machine structures and compiling morphisms. The domains and operations of a machine structure are given and morphisms are characterized as maps preserving both the algebraic structure and the computation structure. A machine structure is defined within the world of tree-structured computation and a morphism from the natural machine structure for sequential computation is defined and proved correct. In Chapter VIII the work presented in this thesis is reviewed and the accomplishments are summarized. Some additional remarks are made on the choice of basic notions and on relations to other work. Further applications and future directions of research are also discussed. Appendix A is a concise and complete definition of the underlying algebraic structure and of the basic computation relations of *Rum*. In addition, some notions that were treated only informally in the main text are given precise definitions. Appendix B contains two substantial examples which illustrate further what we can do in *Rum*. The first example defines a class of derived properties of computation trees and a derivation map on forms such that derived forms compute the derived properties of the computations described by the original forms. The second example treats tree-structured search spaces where trees are given by a successor function and an initial position. A pfn generating a stream of positions according to a given search strategy is defined and key properties proved. We illustrate the use of these tools to define and prove properties of pfns that generate streams by searching.

Guide for reading

The contents of Chapter I and their relevance is explained in more detail in the introduction to that chapter. The reader familiar with Lisp, programming language semantics, program transformations, or recursion theory may wish to skip the corresponding parts of this survey of previous work. The final section of Chapter I should be read as it explains the goals of our work and establishes a framework for understanding some of the choices of basic notions. The illustrated informal introduction to *Rum* (Chapter II) provides a general overview and is recommended reading for all. The introduction to Chapter III explains which parts of the mathematical notation and background presented in that chapter are needed for which parts of the following presentation.

The work on comparison relations (Chapter VI) and the examples in Appendix B are carried out in the world of tree-structured computation and depend only on the definitions and results of Chapter IV. Chapter V extends the world of tree-structured computation and requires knowledge of the material in Chapter IV. The treatment of abstract machine structures and compiling morphisms (Chapter VII) involves both tree-structured and sequential computation.

The examples given in Chapter IV and Appendix B can be understood informally by reading the informal description of the computation primitives (§IV.2) and becoming familiar with notation conventions (§IV.3). (Assuming the illustrated introduction has been read). The work on comparisons and on machines and morphisms is more technical.

To obtain a general perspective of the goals of *Rum* the final section of the background chapter (§I.6) can be read at any time. The review of the work presented in this thesis and of the accomplishments (§VIII.1) can also be read at any time to help the reader form a general picture. For the reader who is puzzled by our choice of basic notions, the discussions in §I.6 and §VIII.2 may be read when such questions arise.

Chapter I. Background

Rum brings together a variety of notions and draws from work in several areas including symbolic computation, program transformations, programming language semantics, and recursion theory. The purpose of this chapter is to introduce the programming tools and informal concepts from current programming practice we wish to treat and to summarize the ideas and results from programming language semantics and logic that are the foundation for our work. Additional remarks on related work can be found in §VIII.3.

Two themes recur. One is the role of application and abstraction in describing computation and in defining functions. The other is the role of program transformations in understanding computations described by programs, deriving programs, and proving properties of programs. Some parts of this chapter will be more meaningful to some people and other parts more meaningful to others, depending on background and interest. The reader should at least be able to get a general idea of each of the areas, and be able learn more by reading suggested references. All of the work discussed went into the consideration and development of *Rum*.

In §1 we explain what we mean by *symbolic computation*, list the goals of a mathematical theory of computation, and give some basic steps towards these goals. This is based on ideas of McCarthy [1960, 1963b].

The work in Lisp and related languages lays a foundation for the practice of symbolic computation and provides many ideas about description of computation, data structures, control structures, and operations on programs. In §2 example programs are given illustrating many of these ideas including simple S-expression recursion, mapping functionals, escape mechanisms, representation of structured data as higher-order objects, streams and co-routines. Variants of these examples will reappear in the *Rum* context where their mathematical properties are studied. This section concludes with a discussion of choices for representation of the computation state in some dialects of Lisp. Consequences of these choices related to the computation primitives provided and the limitations imposed are pointed out. This is to emphasize that a fuller mathematical understanding of the basic computation structures and of the underlying model of computation can be of value in making decisions about the design and implementation of a programming system.

In §3 we look at work involving operations on programs. This work plays an important role in the development of programming tools, compilers, and programming systems. A brief summary of a variety of operations on programs is given. Then the basic ideas underlying systems of program improving transformations (Burstall and Darlington [1976, 1977], Scherlis [1980]) are outlined. Examples are given illustrating the use of program transformations for improving programs and for analyzing improvements by transforming a program into a derived program which computes properties of the computations described by the original program. This work is the starting point for work on transformations in *Rum*. The examples illustrate basic concepts to be generalized to our richer computational model.

§4 concerns interpretations and uses of a simple lambda-calculus-like language AE. Many of the basic ideas underlying the *Rum* model of computation are presented in this section. We begin with an interpretation of expressions of AE given by an abstract machine (SECD) for mechanical evaluation of expressions. Extension of AE to IAE by the addition of imperative constructs and use of IAE as a tool for semantics and language design is also discussed. This is based on ideas of Landin [1964, 1965, 1966]. Next a call-by-value variant of the lambda calculus due to Plotkin [1975] is described. Notions of value and evaluation are compared to those of normal form and reduction; and operational equivalence, a natural equivalence relation on expressions based on evaluation, is compared to lambda equivalence. Further insights concerning application and abstraction as programming tools are obtained by examining the use of AE to define semantic meaning functions for general programming languages. The key ideas are due to Morris and Wadsworth (see Reynolds [1972]). A series of interpreters defined by AE programs is discussed. These illustrate the options for control structure in an interpreter; for representation of objects in the semantic domain; and how these choices affect the relation of the semantics of the defined language to that of the defining language. Of particular interest is the notion of *continuation-passing style* and the existence of machine-like fragments in AE. Interpreters written in this fragment are "absolute" in the sense that the functions defined by programs of the defined language are independent of the choice of evaluation rule for the defining language. The main features of continuation-passing are summarized and some additional work based on transformations into this fragment is discussed.

In §5 we review work in logic related to the construction and analysis of applicative structures sufficiently rich to serve as models of lambda calculus laws for application and abstraction. Of particular interest are theorems expressing key closure conditions satisfied by classes of functions or computations; methods for constructing applicative structures uniformly from given abstract structures; and theorems identifying incompatible sets of requirements for applicative structures. We begin with a discussion of the basic goals of recursion theory. Notions of *a recursion theory* (closure conditions for classes of functions) and of *a computation* *theory* (closure conditions for classes of computations) are introduced as a context for presenting versions of the key theorems – the S^n_m theorem, the recursion theorem and the normal form theorem. Three constructions of applicative structures are presented: recursion on abstract structures (Moschovakis [1969]); extensional models of the lambda calculus (Scott [1976]); and theories of partial operations and classes (Feferman [1975]). Each construction yields a model satisfying the conditions for a recursion theory. The Moschovakis construction is a paradigm for constructing computation theories. Scott models provide independent mathematical constructions of models of the lambda calculus as a language for defining functions. The rich equational theory of these models has served as a paradigm for the study of equations in *Rum*. Feferman's work illustrates a method of extending theories of structures (classes of abstract structures) to intensional theories of partial function on these structures. The extended theories are inconsistent with extensionality. The proof of this fact identifies an important collection of incompatible requirements and suggests that testing for equality on the full computation domain is what must be given up in order to have computationly meaningful intensionality which is consistent with extensionality.

In §6 the goals of *Rum* are explained and we indicate how *Rum* extends and builds on the work summarized in this chapter.

For this chapter we assume that the reader is familiar with the basic notation and notions of the lambda calculus. We use the terms "extensional" and "intensional" to distinguish between the "what" and the "how". For example extensional properties of programs are determined by *what* values are returned, while intensional properties depend on *how* the value is computed. In this framework we can speak of extensional properties of control abstractions as well as of functional abstractions. We caution the reader that we will use the term "function" both in the constructive or intensional sense as a rule for computing a value and in the set theoretic or extensional sense as a graph or set of pairs. Context should make the intended sense clear. We use the term "conditional" to refer to **if-then-else** type computation primitives which express conditional evaluation based on the value of a test subcomputation.

The references given are intended to provide useful pointers to key original papers and to surveys or texts where available. They are not intended as a comprehensive bibliography in any of the areas encompassed.

I.1. About symbolic computation

I.1.1. Symbolic expressions and first-order Lisp

Practical symbolic computation has its origin in the work of McCarthy. The basic ideas are presented in McCarthy [1960], which is the beginning of the language Lisp. Computation is described by symbolic expressions using recursion and conditional evaluation. The data domain is the S-expression domain which is generated by a pairing operation from a set of atomic objects including symbols and numbers. Lists are those S-expressions built from the empty list by pairing an arbitrary S-expression with a list. There is a special atom NIL that represents the empty list. Programs are naturally represented as lists and interpretations of the variable symbols occurring free are represented by *a-lists* (association lists). An a-list is a list of pairs interpreted as associating the first element of each pair with the second. The value of an expression e relative to an interpretation of its free variables a is defined by a computable function on S-expressions $eval(e,a)$. This function was later implemented as a Lisp interpreter (McCarthy et. al. [1962]).

As an example, the function that computes the product of the atoms of an S-expression, using an a-list to interpret symbols is given by the recursive definition

$$\begin{aligned} prod(x,a) \leftarrow \mathbf{if}(&atom(x),\\ &aval(x,a),\\ &prod(car(x),a) * prod(cdr(x),a)) \end{aligned}$$

where *atom* tests for atomic expressions, *car* and *cdr* select the first and second components of a pair, $aval(x,a)$ is the value associated with the atom x by the a-list a. This definition is represented in (Common) Lisp by the S-expression

```
(DEFUN PROD (X A)
       (IF (ATOM X)
           (AVAL X A)
           (* (PROD (CAR X) A)(PROD (CDR X) A)))).
```

S-expressions were designed as a representation of symbolic information that is easy to read, print, and operate on. Some examples of symbolic information are programs, mathematical expressions, logical formulae, rules of inference, schedules, and inventories.

It is important to note that even in full Lisp, the S-expression domain is treated as an "abstract structure" with primitive operations for creating objects, selecting and updating components of composite objects, testing for the identity of two objects and recognizing the different sorts of objects. By abstract structure here we mean that data objects are treated uniformly and can be part of argument lists, bound in environments and returned as values. The size of an object is only of concern when it is necessary to process each sub-object.

§1.2. Basis for a mathematical theory of computation

McCarthy [1963a,b] introduces the notion of a mathematical theory of computation. As this work is the starting point for much of the work in *Rum*, we summarize the main ideas below.

McCarthy [1963b] sets forth goals for a mathematical theory of computation, presents several formalisms for describing computations, and gives some mathematical properties of the formalisms as steps towards achieving these goals. McCarthy's goals are still very relevant and are closely related to the goals of *Rum*. They are (in abbreviated form)

1. To develop a universal programming language.
2. To define a theory of equivalence of computation processes. This would be the basis for a theory of equivalence preserving transformations.
3. To represent algorithms by symbolic expressions in such a way that significant changes in the behavior represented by the algorithms are represented by simple changes in the symbolic expressions.
4. To represent computers as well as computations in a formalism that permits a treatment of the relation between a computation and the computer that carries out the computation.
5. To give a quantitative theory of computation. For example to find a quantitative measure of the size of a computation analogous to Shannon's measure of information.

The key formalism given by McCarthy for describing computations is the definition of a class of computable functions over a given data domain and a given set of operations on that domain by systems of recursion equations. A system of recursion equations has the form

$$(\leftarrow 1) \qquad f_1(x_1,\ldots,x_{n_1}) \leftarrow e_1$$

$$\ldots$$

$$(\leftarrow k) \qquad f_k(x_1,\ldots,x_{n_k}) \leftarrow e_k$$

where for $1 \leq j \leq k$, f_j is a function symbol of arity n_j and e_j is an expression built from the variables $x_1,\ldots,x_{n_j}$, constants for data and given operations, and the symbols f_i, $1 \leq i \leq k$, using application and conditional (**if-then-else**). $(\leftarrow j)$ is the defining equation for f_j and the definitions are used recursively to compute the value of any such expression relative to an assignment of values in the data domain to the variable symbols of the expression. The value of an **if-then-else** expression is computed by computing the value of the **if** subexpression. If the result is true the **then** subexpression is evaluated otherwise the **else** subexpression is evaluated. In either case the other branch subexpression is not evaluated. Values of argument expressions must be computed before a given or defined function can be applied. Defined functions are applied by evaluating the right-hand side of the defining equation where the values of the argument expressions are assigned to the variables of the left-hand side. This formalism provides an interpretation of systems of recursion equations both as rules of computation and as partial functions. Thus it provides an interpretation of pure Lisp-like programs as partial functions.

Two key mathematical results given by McCarthy, about the above formalism for describing computation, are:

- a notion of equivalence for conditional expressions and axioms and rules sufficient for transforming an expression to any equivalent expression
- The principle of ***recursion induction*** for proving equations involving recursively defined functions - let f be a recursively defined function and let g any function with the same domain of definition as f which satisfies the defining equation for f. Then g and f are the same partial function.

A further important idea, introduced in McCarthy [1963a] is *abstract syntax*. The abstract syntax of a language specifies the set of expressions and operations for synthesis and analysis abstractly, independent of any notation. It is defined by giving a signature and axioms. The signature consists of names for constructors of expressions (the synthetic part), and for operations recognizing the various constructions and selecting components (the analytic part). The axioms specify the relations among the constructors, selectors and recognizers.

Remarks

• The interpretation of systems of recursions equations as rules is used as a basis for operational reasoning about properties of the partial functions computed. Recursion induction is an example. This interpretation is not developed to provide a basis for representing and proving intensional properties.

• The functions computed by a system of recursion equations are the least fixed points of the corresponding system of functionals $<\tau_1 \ldots \tau_k>$ where

$$\tau_1 = \lambda(f_1 \ldots f_k)\lambda(x_1, \ldots, x_{n_1})e_1$$

$$\ldots$$

$$\tau_k = \lambda(f_1 \ldots f_k)\lambda(x_1, \ldots, x_{n_k})e_k.$$

This is a form of inductive definition of partial functions. The principle of recursion induction is a consequence of the minimality of the defined functions. (See

Moschovakis [1975] for a general treatment of partial functions defined by such systems of functionals.)

- A modern version of abstract syntax is the notion of initial algebra (Goguen and Meseguer [1983]) extended by operations for recognizing and selecting.

I.2. Programming examples

Early work on the lambda calculus (Kleene [1936a], Church [1941]) established the expressive power of abstraction and application, interpreted using lambda reduction rules, for defining functions. Central in this work was the use of functions as values to represent definition mechanisms such as recursion, iteration, and minimization, and to represent structured objects such as tuples. Early examples of programming using function and control abstractions are given in Burstall [1968] and Burge [1971]. Many more examples can be found in Sussman and Steele [1975], Friedman et. al. [1984], and Abelson and Sussman [1985]. The examples in this section illustrate some of the key features of such programs.

I.2.1. Notation

S-expressions will be used as the data domain for examples in this chapter. x, y, z, ... range over S-expressions and we use standard notation for numbers and arithmetic expressions. S-expression symbol constants are upper case identifiers in THIS FONT, for example NIL, A, B, and CAR denote symbolic atoms. We use $x \bullet y$ as infix notation for the pairing operation $cons(x, y)$, $\texttt{<}y_1, \ldots, y_n\texttt{>}$ for the multi-ary list forming operation, and $x \diamond y$ as infix notation for *append* (list concatenation). Thus $\texttt{B} \bullet \texttt{NIL}$ and <B> are expressions denoting the list with a single element B and <<B>> is a list with one element, the list <B>, which in turn has one element, the symbol B. Using the associativity of append and letting $\bullet$ associate to the right we have

$$\texttt{<A,<<B>>,C>} = \texttt{A} \bullet ((\texttt{B} \bullet \texttt{NIL}) \bullet \texttt{NIL}) \bullet \texttt{C} \bullet \texttt{NIL} = \texttt{<A>} \diamond \texttt{<<<B>>>} \diamond \texttt{<C>}.$$

In some of the examples we also use the traditional Lisp notation for list constants as illustrated by the S-expression representation of the *prod* program above. In general, () represents the empty list. ((B)) = <<B>> and <A,<<B>>,C> = (A ((B)) C). Use of Lisp list notation for list constants in mathematical expressions may lead to confusion since parentheses might be merely grouping symbols or they might denote a level of list formation. We will use the Lisp notation only where such confusion can not arise.

Finite sequences of elements a_i are written $[a_0, \ldots a_n]$. □ is the empty sequence. $[u, v]$ is the concatenation of sequences u and v. Individual elements are also treated as sequences of one element. For conditional expressions, we write $\mathtt{if}(e_0, e_1, e_2)$ rather than **if** e_0 **then** e_1 **else** e_2.

We use lambda notation for function expressions. For example, $\lambda(z)\mathtt{A}$ is the constant function with value A and $\lambda(x,y)x(y)$ is a function of two arguments that applies its first argument to its second argument. Braces are often used to enclose complex expressions in function positions to improve readability. Thus e_0 applied to e_1 may be written $\{e_0\}(e_1)$ and we have

$$\{\lambda(x,y)x(y)\}(car, (\mathtt{A}\ \ \mathtt{B})) = \mathtt{A}.$$

We will have occasion to use two notions of substitution of one expression into another. $e|^x_{e_1}$ denotes the result of substituting e_1 for free occurrences of x in e, renaming bound variables of e if necessary to avoid trapping free variables of e_1. This is the usual logical notion of substitution. The other notion is that of placing an expression into an expression context. An expression context is an expression e with a hole to be filled (written $e\{\ldots\}$). $e\{e_0\}$ is the result of putting the expression e_0 in the hole, without any renaming of bound variables in e. (This is a second use of braces, but should cause no confusion.) One can think of an expression context as an expression containing a free occurrence of a distinguished variable symbol marking the hole.

For expressions that may not have a value we write $e_0 \sim e_1$ to mean that either both e_0 and e_1 are undefined or both are defined with the same value. In case both expressions are known to be defined we may write $e_0 = e_1$.

I.2.2. Mapping functionals

A common operation in symbolic computation is to "map" through a list or tree structure, applying a given function at each point and collecting the results using an operation such as *cons* to collect into a list or + to collect into a number. Such functionals are given by special kinds of schemes for recursion on the argument structure and are implemented by programs traditionally called mapping functions in Lisp. For example *mapcar* takes a function and a list as arguments, applies the function to each element of the list, and returns a list of the results. Thus if *tack* is defined by

$$tack(x,y) \leftarrow mapcar(\lambda(z)cons(x,z), y)$$

then

$$tack(x, \mathtt{<}\ldots, y_i, \ldots\mathtt{>}) \sim \mathtt{<}\ldots, (x \bullet y_i), \ldots\mathtt{>}$$

i.e. the S-expression x is tacked to each member of the list y. Mapping functions are distributive functionals – the mapping operation distributes over suitable operations on the recursion argument. They satisfy a rich collection of algebraic laws derived from laws for operations on the argument and value structures. For example for lists y, z

$$tack(x, y \diamond z) = tack(x, y) \diamond tack(x, z).$$

I.2.3. Escape mechanisms

Escape mechanisms are important programming tools. Error handling is based on such mechanisms. They also provide a way of pruning unnecessary computation and allow certain computations to be expressed by more compact and conceptually manageable programs. The idea is to mark a given point in a computation and to be able to continue from that point when special conditions arise in the course of a computation rather than continuing in the normal way. For example suppose p computes a predicate on S-expressions. We would like to search an S-expression for a subexpression satisfying the predicate and to terminate the search as soon as such a subexpression is found. If no such subexpression is found, `NIL` is returned. *find* carries out such a search.

```
$find(x,p) \leftarrow$ catch(TAG, $findit(x,p)$)
$findit(x,p) \leftarrow$ if($p(x)$,
                     throw(TAG, $x$)
                     if($atom(x)$,
                        NIL,
                        progn[$findit(car(x),p)$,
                              $findit(cdr(x),p)$])),
```

catch, **throw**, and **progn** are traditional Lisp names for program constructs. **catch**(TAG, e_0) is used to mark the point at which computation of e_0 begins with the symbol TAG. If during the evaluation of e_0 an expression **throw**(TAG, e_1) is executed then e_1 is evaluated and the value of e_1 is returned as the value of the **catch** expression. That is, computation resumes at the point marked by TAG returning the value of e_1. If the evaluation of e_0 returns a value normally then that value is returned by the **catch** expression. **progn** evaluates a sequence of expressions and returns the value of the last one.

I.2.4. Structured data represented as functions

The representation of structured objects as functions provides a uniform mechanism for data representation in terms of well understood concepts of function application and abstraction. These ideas go back to early work in the lambda calculus (Kleene [1936], Church [1941]) and were proposed in the context of programming by Reynolds [1970]. We will illustrate two representations of a pairing structure. The basic ideas generalize to a wide class of abstract data structures.

lcons, *lcar*, *lcdr* correspond to the usual representation of a pairing structure in the lambda calculus.

$$\begin{aligned}
&\mathit{lcons} \leftarrow \lambda(x,y)\lambda(z)z(x,y)\\
&\mathit{lcar} \leftarrow \lambda(z)z(\lambda(x,y)x)\\
&\mathit{lcdr} \leftarrow \lambda(z)z(\lambda(x,y)y)
\end{aligned}$$

Thinking of these as definitions in lambda calculus *lcons*, *lcar*, *lcdr* obey the usual the pairing laws. For example,

$$\mathit{lcar}(\mathit{lcons}(x,y)) \equiv \{\lambda(z)z(x,y)\}\lambda(x,y)x \equiv \{\lambda(x,y)x\}(x,y) \equiv x.$$

Thinking of the above definitions as defining objects, *lcons*(A, D) is an object which associates A with x and D with y and we have

$$\mathit{lcar}(\mathit{lcons}(\mathtt{A},\mathtt{D})) \sim \mathtt{A}$$

ocons, *ocar*, *osetcar*, *opairp* represent (part of) a pairing structure in the *object oriented* style of programming à la Small-talk (Goldberg and Robson [1983]) or Actors (Hewitt [1977]). In this style, objects contain the information about the operations that can be applied and are sent messages naming the operation to be carried out. An object also may have an internal state which can be updated by the object. In particular, the object oriented representation extends the notion of pairing structure to include updating operations.

$$\begin{aligned}
\mathit{ocons} \leftarrow \lambda(x,y)\lambda(\mathit{msg})&\mathbf{if}(\mathit{msg} = \mathtt{PAIR?}), \mathtt{T}\\
&\mathbf{if}(\mathit{msg} = \mathtt{INTEGER?}), \mathtt{NIL}\\
&\mathbf{if}(\mathit{msg} = \mathtt{CAR}), x\\
&\mathbf{if}(\mathit{msg} = \mathtt{CDR}), y\\
&\mathbf{if}(\mathit{msg} = \mathtt{SETCAR}), \lambda(z)[x \leftarrow z],\\
&\mathbf{if}(\mathit{msg} = \mathtt{SETCDR}), \lambda(z)[y \leftarrow z],\\
&\qquad\qquad\qquad \ldots\\
&))))) \ldots)
\end{aligned}$$

$$\begin{aligned}
&\mathit{opairp} \leftarrow \lambda(z)[z(\mathtt{PAIR?})]\\
&\mathit{ocar} \leftarrow \lambda(z)[z(\mathtt{CAR})]\\
&\mathit{osetcar} \leftarrow \lambda(z,x)\{z(\mathtt{SETCAR})\}(x)
\end{aligned}$$

ocons creates pairs, *ocar* gets the first component, *osetcar* sets the first component, and *opairp* tests for pairs (among a collection of similarly represented data types).

Executing *ocons*(A, D) creates an object p which has internal state associating A to x and D to y. Executing *ocar*(p) causes p to be applied to CAR (in object jargon, the message CAR is sent to p). p responds by returning the value currently associated to x. If no updating messages have been received by p since its creation, then p will respond to the message CAR with A. *osetcar*(p, B) sends p the message SETCAR. p responds with a function object q with access to the internal state of p. When q is applied to an argument, p's internal store is updated to associate that argument to x. Thus if *osetcar*(p, B) is executed then p will respond to the message CAR with B. p will always respond to the PAIR? message with T, and to the INTEGER? message with NIL.

I.2.5. Streams

Streams are a means of presenting elements of a possibly infinite sequence. The elements are accessed sequentially by request for the next element. A stream may be implemented to generate the next element when requested rather than in advance. Typically a stream s is a 0-ary function-like object with internal state. Request for the next element is application to the empty argument list, $s()$. The source of stream elements and whether or not they are computed in advance or upon request is invisible to the user of the stream.

Streams are an important tool for input and output. Character streams are a typical example. If a reader or parser has a character stream as a parameter, then the same program can be used to read from a file, a tape, an array, a list, or a terminal by constructing the appropriate stream. For example, the program *record.to.char* is a character stream that provides an interface to an input device that produces 80 character records. The device could be a card reader, a disk file, a terminal, etc. The statement **stream**$\{a[1:80], n \leftarrow 80\}$ says that what is created is a stream object with internal store a, n. a is a tuple of length 80 and n is a number, initially 80. Executing $a \leftarrow$ *read.record*() reads the next record from the device into the tuple a. $nth(a, n)$ is the n-th element of a.

```
record.to.char ←stream{a[1 : 80], n ← 80}
                       if(n < 80,
                          progn[n ← n + 1,
                                nth(a, n)],
                          progn[a ← read.record(),
                                n ← 1,
                                nth(a, n)])
```

At any stage in processing input from the device, the internal state of *record.to.char* is such that a contains the current record and n indexes the last character read.

When $n < 80$ the next character is the $n + 1$-st element of a. When $n > 80$ the next character is the first character of the next record.

Another use of streams is to iterate along a sequence of items given by a sequence of expressions, where the expressions are not to be evaluated until needed. This idea was introduced by Landin [1965] to interpret certain iteration constructs of Algol60.

I.2.6. Co-routines

Another useful control mechanism is co-routines. The idea was originally presented in Conway [1963] as a means of separating a complex compiling program into a number of small independent procedures. Each procedure is a co-routine responsible for some phase in the transformation of a source program represented as a sequence of characters through various intermediate stages to the final stage – the compiled code. In a system of co-routines control is transferred from one co-routine to another by *resuming* rather than calling. When one co-routine resumes another, it remembers the point where it left off and begins at that point when control is transferred back to it, i.e. when it is resumed by a partner co-routine. The classical example (given by Conway) is *squasher* which transforms a stream of characters by "squashing" adjacent pairs of * characters (Fortran for exponent) to a single ↑ character (Algol for exponent). The statement **co-routine**$\{x, y\}$ expresses that *squasher* is a co-routine with internal state components named by x, y. There is also an unnamed component of the internal state for remembering where to resume. The body of *squasher* is a sequence of instructions. The instruction $x \leftarrow readch()$ assigns to x the next character from the input. **ifeq**$(e_0, e_1); e$ executes e only if e_0 and e_1 have the same value. *user* is the partner co-routine. Each time *user* wants another character it resumes *squasher* and will be resumed by *squasher* when the next character is determined.

```
$squasher$ ←co-routine$\{x, y\}$
                SQ : $x \leftarrow readch()$
                S0 : ifeq$(x, *)$; goto S1
                     resume$(user, x)$
                     goto SQ
                S1 : $y \leftarrow readch()$
                     ifeq$(y, *)$; goto S3
                     resume$(user, x)$
                     resume$(user, y)$
                     goto SQ
                S3 : resume$(user, \uparrow)$
                     goto SQ
```

Initially *squasher* resumes at the instruction labeled SQ and x is assigned the next character from the input. If $x \neq *$ then *user* is resumed passing it x and when *squasher* is next resumed the **goto** SQ instruction will be executed. If $x = *$ then y is assigned the next character. If $y \neq *$ then *user* is resumed with x and when *squasher* is next resumed it will execute the **resume**$(user, y)$ instruction. If $y = *$ then *user* is resumed with $\uparrow$ and when *squasher* is next resumed the **goto** SQ instruction will be executed.

I.2.7. Features of Lisp systems

Lisp systems are a means for interactively developing programming tools, building data structures, and for carrying out computations using the tools developed. A Lisp system has internal state that assigns values and other information to symbols. Interaction is carried out by evaluating symbolic expressions. The values assigned by the internal state provide the initial evaluation environment. Evaluation may simply return a value (answer a question). It may also modify the internal state by adding to or updating the information stored there. At any point in the process of evaluation there are three key components of the computation state: the expression currently being evaluated, the current evaluation environment, and the evaluation context for the current expression – what is to be done with its value. We now examine implementation of these features in more detail for several Lisp systems. We will indicate, for some of the computations discussed above, how they can be represented in these systems or why they cannot be represented (directly).

Lisp 1.5

In Lisp 1.5 (McCarthy et. al. [1962]) environments were represented by a-lists as in McCarthy's original description of Lisp. Lambda expressions could appear in the function position of an expression. In such cases they were not evaluated, but served as a way of temporarily naming the values of argument expressions. The **function** construct was introduced to provide a means of lexically fixing the values of the free variables of lambda expressions used as functional arguments. The value of **function**$(\lambda(x)e)$ is a *funarg* object containing the lambda expression and the current environment. When a funarg is applied, the environment component is used to interpret the free variables of the lambda expression. Funargs were first class objects and provided a means of returning functions as values. For example, in Lisp 1.5 the definition of *lcons* is written

$$lcons \leftarrow \lambda(x,y)\mathbf{function}(\lambda(z)z(x,y)).$$

However, not much use was made of this feature of Lisp 1.5.

Error trapping mechanisms were provided in Lisp 1.5 in order to give system builders control over what happened when an error occurred. These primitives were discovered to be useful for programming more general escape mechanisms such as the *findit* example. To escape with a value using the error mechanism, it was necessary to use side effects - for example setting the value of some global parameter.

Maclisp

In Maclisp (Pitman [1983]) the computation state is a stack structure. A function call pushes control and variable binding information on the stack. When a value is returned this information is popped (deleted) from the stack and used to resume computation in the calling context. Maclisp provides funargs, but the environment component of a funarg object in Maclisp is a "binding pointer" that refers to the stack position where the binding information is stored. This creates what is called the "funarg problem" (Moses, [1970]), although in fact it is the "funval problem". The problem is that a funarg is not an object with an independent existence. It is only meaningful to apply it within the dynamic scope of the environment (function call) in which it was created. After the computation control exits that scope the binding pointer of a funarg no longer refers to the intended variable bindings. Thus funargs can not be used in Maclisp to implement *lcons* or *ocons.*

In order to separate program control mechanisms from error mechanisms, **catch** and **throw** primitives were introduced in Maclisp. The primitives used in the *findit* program above are based on the Maclisp primitives. In the simple case, the **catch** tag parameter is used to mark the control stack and the **throw** tag parameter is used to determine the point where the computation is to be continued by searching the control stack for the matching mark. Like funargs, catch tags are only meaningful with in the dynamic scope of their defining context, and a catch tag cannot be thrown to after popping the control stack beyond the point where the tag was defined. Thus **catch** and **throw** are not adequate to implement co-routines.

Interlisp spaghetti

In Interlisp, data structures called *environment descriptors* are used to represent the state of a computation (Bobrow and Wegbreit [1973]). These structures are implemented as stack-like objects called "spaghetti stacks" which contain additional linkages between stack positions and elements of the stack. [Pointer diagrams for these structures look rather like spaghetti.] Environment descriptors are first class objects which may be assigned to variables, passed as arguments and returned as values. There are operations for creating and updating environment descriptors and for selection of control and variable binding components of environment descriptors. Environment descriptors can be used to implement function abstractions such as funargs and control abstractions such as **catch** and **throw**, and co-routines.

Scheme

Scheme (Sussman and Steele [1975]) is the first dialect of Lisp to take abstraction and application seriously. The main components of a Scheme computation state are an expression, an environment and a continuation. A continuation is a function of one argument representing the calling context for the expression component. The function position of an application is evaluated as any other subexpression. The value of a lambda-expression is a *closure* containing the expression and the evaluation environment (Scheme closures behave like funargs of Lisp 1.5). The **catch** construct of Scheme generalizes that of Maclisp by binding the current continuation to a variable rather than putting a mark in the continuation. Closures and continuations are first class objects and give much of the capability of Interlisp environment descriptors. The Scheme model of computation derives from work of Landin (see §4).

In addition to Lisp dialects, a number of other programming systems provide higher level abstractions. Of principal interest to our work are ISWIM (Landin [1966] and Burge [1981]), Pop-2 (Burstall and Popplestone [1968]), and ML (Milner [1984]).

For more details on the history and development of Lisp see McCarthy [1978]. The current state of Lisp is described in (Steele [1984]).

I.3. Program transformations

There are a variety of operations on programs carried out for a variety of purposes –

(i) converting programs annotated with assertions about the computation state at the annotated points into logical formulae – assertion methods of verification (Manna [1969])

(ii) annotating programs with information about type, control and data flow, etc. – as an aid to compiling and program understanding (Steele [1978])

(iii) transforming from one language to another or to a fragment of the given language – compiling (Steele [1978])

(iv) using automatic deduction to derive defining equations from specifications – program synthesis (Manna and Waldinger [1980])

(v) using proof normalization to construct programs and using programs annotated with proofs (obtained from the normalization process) to construct optimized programs for special cases (Goad [1980])

(vi) transformations changing the structure of the computation described while preserving the function computed – program optimization (Burstall and Darlington [1976, 1977], Scherlis [1980])

(vii) transforming a program into a program that computes a property of computation described by the original program – derived programs (Wegbreit [1975], McCarthy – private communication)

In the following we will focus on transformations such as (vi) and (vii) and restrict attention to programs given by systems of recursive definitions over a given data structure (see §1).

I.3.1. Improving programs

Burstall and Darlington [1976, 1977] studied transformation rules with the goal of developing systems for automatically improving programs. They identified four improvements: (i) recursion elimination (transforming recursion into iteration), common subexpression elimination (abstraction), procedure call elimination (application), and reuse of storage. The basic idea is to begin with a system of definitions $\mathcal{E}$ containing exactly one definition for each defined function symbol. New definitions are added according to a set of transformation rules, and a newly derived definition may be taken as *the* definition (rule for computation) of its function symbol. Some typical transformation rules are

- (instantiation) If $f(x_1,\ldots,x_n,y,z_1,\ldots,z_m) \leftarrow e$ is in $\mathcal{E}$ and l is a constant term then add $f(x_1,\ldots,x_n,l,z_1,\ldots,z_m) \leftarrow e'$ where e' is the result of replacing y by l in e.
- (unfolding) If $f_a(x_1,\ldots,x_n) \leftarrow e_a$ and $f_b(y_1,\ldots,y_m) \leftarrow e_b$ are in $\mathcal{E}$ and e_c is obtained from e_a by replacing an instance of $f_b(y_1,\ldots,y_m)$ by the corresponding instance of e_b then add $f_a(x_1,\ldots,x_n) \leftarrow e_c$.
- (folding) If $f_a(x_1,\ldots,x_n) \leftarrow e_a$ and $f_b(y_1,\ldots,y_m) \leftarrow e_b$ are in $\mathcal{E}$ and e_c is obtained from e_a by replacing an instance of e_b by the corresponding instance of $f_b(y_1,\ldots,y_m)$ then add $f_a(x_1,\ldots,x_n) \leftarrow e_c$.
- (simplification) If $f_a(x_1,\ldots,x_n) \leftarrow e_a$ and e_a simplifies to e_c according basic laws (for data operations, conditional, etc.) then add $f_a(x_1,\ldots,x_n) \leftarrow e_c$
- (definition) If f_{new} is function symbol not mentioned in $\mathcal{E}$ and the variables of e are among $x_1,\ldots,x_n$ then add $f_{new}(x_1,\ldots,x_n) \leftarrow e$.

The function computed by a definition in the initial system and the function computed by a definition in a system derived by these rules used without restrictions will agree on their common domain of definition, but they may have different domains of definition.[1] One problem is that arguments to functions may not be used in the defining expression, thus an undefined term may disappear upon unfolding. Another problem is that using folding it is easy to derive trivial definitions such as $f(x_1, \ldots, x_n) \leftarrow f(x_1, \ldots, x_n)$. Thus there is additional work to be done after an arbitrary sequence of transformations to verify that domains of definition have been preserved. This is not very satisfactory if the rules are to be used as a formal system.

I.3.2. Transformations on expression procedures

Various restricted systems of rules have been proposed that give rise to transformations that preserve domains of definition. Such transformation rules are called *safe*. Scherlis [1980] describes a transformation system for expression procedures. In this system transformations are carried out on a system of definitions of the form $E \leftarrow F$ where E and F are expressions. Such definitions are used as rules for computing by matching the left hand side and replacing it by the corresponding righthand side – thus treating expressions as complex function names. The key transformation rules are

- (apply) If $E \leftarrow H\{F(e_1, \ldots, e_m)\}$ and $F(x_1, \ldots, x_m) \leftarrow G(x_1, \ldots, x_m)$ are in $\mathcal{E}$ then add $E \leftarrow H\{G(e_1, \ldots, e_m)\}$.
- (compose) If $E \leftarrow F$ is in $\mathcal{E}$ then add $H\{E\} \leftarrow H\{F\}$.
- (abstract) If $E \leftarrow H\{G(e_1, \ldots, e_m)\}$ is in $\mathcal{E}$ then add $E \leftarrow H\{f(e_1, \ldots, e_m)\}$ and $f(x_1, \ldots, x_m) \leftarrow G(x_1, \ldots, x_m)$ where f is a new function symbol.
- (simplify) like the Burstall and Darlington rule.

where $H\{\ldots\}$ is an expression context. The composition rule provides a means of specializing to a particular context and thus to make improvements based on this additional information that would not be valid in general. Restrictions on the transformation rules that insure safety are defined by introducing a notion of computational progress. For a given initial system of definitions, computational progress is defined by finding a suitable well-founded relation on expressions. A definition $E \leftarrow F$ is progressive relative to a system of definitions if computational progress is made whenever a ground instance of E is replaced by the corresponding

[1] Burstall and Darlington worked with systems of equations interpreted using a call-by-name computation rule rather than call-by-value. The general ideas are the same and the same problems just appear in different guises.

instance of F. The initial system is progressive, by criteria imposed on the well-founded relation, and the rules are restricted to a set that preserve progressiveness. Thus safeness is insured. This system is more expressive than the original system of Burstall and Darlington and can in fact be used as a theorem proving system deriving such theorems as associativity of *append*.

To illustrate these rules we derive an iterative definition of the function that reverses lists from a simple recursive definition. For this derivation we assume that the data domain consists of lists and that *append* is a primitive operation satisfying the following axioms (simplification rules).

(if.null) $(u = \mathtt{NIL} \rightarrow e_a = e_b) \rightarrow \mathbf{if}(null(u), e_a, z) = \mathbf{if}(null(u), e_b, z)$

(app.if.dist) $\mathbf{if}(x, y, z) \diamond v = \mathbf{if}(x, y \diamond v, z \diamond v)$

(app.assoc) $(u \diamond v) \diamond w = u \diamond (v \diamond w)$

(app.nil) $\mathtt{NIL} \diamond u = u \diamond \mathtt{NIL} = u$

(app.one) $cons(u, \mathtt{NIL}) \diamond v = cons(u, v)$

The derivation begins with a system consisting of the single definition (revr) for reverse.

(revr) $revr(u) \leftarrow \mathbf{if}(null(u), u, revr(cdr(u)) \diamond cons(car(u), \mathtt{NIL}))$

Using composition, (revr.v) is added, defining the expression procedure $revr(u) \diamond v$. Note that the recursive call to *revr* is an instance of this expression.

(revr.v) $revr(u) \diamond v \leftarrow (\mathbf{if}(null(u), u, revr(cdr(u)) \diamond cons(car(u), \mathtt{NIL}))) \diamond v$

By the simplification rule (revr.v) is replaced by (revr.v′) using (if.null), (app.if.dist), (app.nil) and (app.one).

(revr.v′) $revr(u) \diamond v \leftarrow \mathbf{if}(null(u), v, revr(cdr(u)) \diamond cons(car(u), v))$

Applying abstraction to (revr.v′), (revr.rev) and (rev.revr) are added, introducing *rev* as a name for the procedure $\lambda(u, v) revr(u) \diamond v$.

(revr.rev) $revr(u) \diamond v \leftarrow rev(u, v)$

(rev.revr) $rev(u, v) \leftarrow \mathbf{if}(null(u), v, revr(cdr(u)) \diamond cons(car(u), v))$

Instantiating (revr.rev) with $v = \mathtt{NIL}$ and using (app.nil) we obtain (reverse).

(reverse) $revr(u) \leftarrow rev(u, \mathtt{NIL})$

Finally using application of (revr.rev) in (rev.revr) we obtain (rev)

(rev) $rev(u, v) \leftarrow \mathbf{if}(null(u), v, rev(cdr(u)), cons(car(u), v))$

Taking (reverse,rev) as the final system of equations we have the improved definition of *revr*.

I.3.3. Derived programs

Another kind of program transformation is the definition of operations that change a program into one that computes some property of the computation described by the original program. The resulting programs are called *derived* programs by McCarthy. The effect of such a derivation is to transform intensional properties of a given program into extensional properties of the derived program. For example

$$cons.rev(x) \leftarrow \text{if}(null(x), 0, 1 + cons.rev(cdr(x)))$$

is derived from the definition of *revr* given above by the system of equations (reverse, rev) and computes number of *cons* operations done in reversing a list using this definition. It is easy to see that

$$cons.rev(x) = length(x)$$

Using $cons.append(x,y) = length(x)$, a similar analysis of the definition of *revr* given by the equation (revr) above gives

$$cons.revr(x) = (length(x) * (length(x) + 1))/2 = \sum_{i=0}^{length(x)} i$$

This provides a measure of the improvement produced by the program transformation.

Wegbreit [1975] gives a system for mechanical analysis of the cost of executing programs - considering such costs as procedure call, variable references, and execution of primitive operations. The analysis produces a program for computing the cost of executing a given program as a function of its input parameters. Scherlis's system has been extended to carry out this sort of transformation. This is accomplished by the addition of rule schemes for each program construct and for the costs to be measured (Scherlis, private communication). In addition to costs associated with computations, one may also be interested the maximum amount of a resource in use at any point in a computation. For example, if we elaborate our computation model so that *cons* is a storage allocation primitive, then a derivation can be defined such the derivation of a program *p* computes the maximum amount of storage in use at any point in the computation described by *p*.

There are more exotic derived programs, for example *trace.rev*(x, y) computes a "trace" of the computation of *rev*(x, y), i.e. a tree of instances of function calls.

```
trace.rev(x,y) ← <<REV,x,y>,
                  <NULL,x>,
                  if(null(x),
                     NIL,
                     trace.rev(cdr(x),cons(car(x),y)))>
```

For the example,

```
trace.rev((1 2), NIL) = ((REV (1 2) NIL)
                          (NULL (1 2))
                          ((REV (2) (1))
                           (NULL (2))
                           ((REV NIL (2 1))
                            (NULL NIL))))
```

I.4. Applicative expressions

I.4.1. Mechanical evaluation of applicative expressions

Landin's work developing the theory of applicative expressions as rules for computation and as a place to represent programming concepts is the starting point of our discussion and the source of several important ideas. Landin [1964] considers the problem of assigning 'applicative' structure to familiar mathematical expressions in order to be able to infer rules for computation of the values denoted by such expressions. A lambda-calculus-like language AE (for Applicative Expressions) is defined together with a computation domain and an abstract machine SECD for "mechanical evaluation" of expressions of AE. Expressions of AE are built from variable symbols by application $e_0(e_1)$ and abstraction $\lambda(x)e$. Objects called ***environments*** assign values in the computation domain to free symbols of expressions. The value of $\lambda(x)e$ in an environment ξ is a *closure* $\langle\lambda(x)e \mid \xi\rangle$, which is the function determined by $\lambda(x)e$ with the free symbols fixed to be the values assigned by ξ. The computation domain (values) consists of given data, primitive functions, and closures. The rules for computation are uniformly parameterized by the given data and primitive functions by assuming there is also given a binary application function *ap* which interprets the primitive functions. $ap(f,v)$ is the result of applying the primitive function f to the value v. A primitive function may be applied to any sort of value - data, another primitive function or a closure, and may return any sort of value. [This allows conditional to be treated as a primitive function, and provides for binary primitive functions by currying.]

The SECD machine is a set of states together with a set of transition rules. An SECD machine state is a tuple $<\sigma,\xi,\gamma,\delta>$. σ is the value stack (S), a sequence of intermediate values for the current subcomputation. γ is the control stack (C), a sequence of expressions to be evaluated and application marks *ap* which serve to group pairs of values which come from an application expression. ξ is an environment (E) used to interpret free variables occurring in expressions on the control stack. δ is a dump (D), which is either empty or a state $<\sigma',\xi',\sigma',\delta'>$. The dump describes what is to be done when the control stack is exhausted, i.e.

when evaluation of the current subexpression is complete. The dump represents the computation context built up when the current subcomputation was begun. The set of transition rules for the SECD machine is so short, we list it in entirety.

$$\begin{array}{ll}
\text{(var)} & <\sigma,\ \xi,\ [\llbracket x\rrbracket,\gamma],\ \delta> \ \mapsto\ <[\xi(\llbracket x\rrbracket),\sigma],\ \xi,\ \gamma,\ \delta> \\
\text{(lam)} & <\sigma,\ \xi,\ [\llbracket\lambda(x)e\rrbracket,\gamma],\ \delta> \ \mapsto\ <[\langle\lambda(x)e \mid \xi\rangle,\sigma],\ \xi,\ \gamma,\ \delta> \\
\text{(app)} & <\sigma,\ \xi,\ [\llbracket e_0(e_1)\rrbracket,\gamma],\ \delta> \ \mapsto\ <\sigma,\ \xi,\ [\llbracket e_1\rrbracket,\llbracket e_0\rrbracket,ap,\gamma],\ \delta> \\
\text{(ap.clos)} & <[\langle\lambda(x)e \mid \xi_0\rangle,v,\sigma],\ \xi,\ [ap,\gamma],\ \delta> \ \mapsto \\
 & \qquad\qquad <\square,\ \xi_0\{\llbracket x\rrbracket \leftarrow v\},\ \llbracket e\rrbracket,\ <\sigma,\xi,\gamma,\delta>> \\
\text{(ap.prim)} & <[f,v,\sigma],\ \xi,\ [ap,\gamma],\ \delta> \ \mapsto\ <[ap(f,v),\sigma],\ \xi,\ \gamma,\ \delta> \\
\text{(ret)} & <[v,\sigma],\ \xi,\ \square,\ <\sigma_0,\xi_0,\gamma_0,\delta_0>> \ \mapsto\ <[v,\sigma_0],\ \xi_0,\ \gamma_0,\ \delta_0>
\end{array}$$

Here, following the conventions of work in semantics, we are using $\llbracket e\rrbracket$ to emphasize that we are thinking of e as a data structure (in some suitable encoding). $\xi(\llbracket x\rrbracket)$ is the value associated to the variable $\llbracket x\rrbracket$ by the environment ξ and $\xi_0\{\llbracket x\rrbracket \leftarrow v\}$ is the environment that associates v to $\llbracket x\rrbracket$ and is otherwise like ξ_0.

To compute the value of an expression e in an environment ξ the SECD machine is started in the state $<\square,\xi,\llbracket e\rrbracket,\square>$. Computation proceeds by repeated application of the transition rules. The computation returns the value v if a state $<v,\xi,\square,\square>$ is reached.

Note that the structure of an SECD machine state amounts to using a stack to implement a simple recursive definition of the value of an expression environment pair. (See our discussion of Reynolds [1972] below.)

Landin [1965] extends AE to IAE (imperative applicative expressions) by adding imperative computation primitives. The primitive of interest here is the [in]famous "Landin J operator". $J(\lambda(x)e)$ is a *program point* – it marks a point in the execution of program. The value of a program point is a *program closure* which contains the lambda-expression, the current environment, and the current dump. Application of a program closure applies the lambda-closure part to the argument and returns the value at the point where the program closure was made. The rule for application of a program closure is

$$<[\langle J(\lambda(x)e) \mid \xi_0,\delta_0\rangle,v,\sigma],\xi,[ap,\gamma],\delta> \ \mapsto\ <\square,\xi_0\{\llbracket x\rrbracket \leftarrow v\},\llbracket e\rrbracket,\delta_0>$$

IAE was used by Landin to interpret Algol60 programs and hence to provide a simple and computationally meaningful semantics.

Landin [1966] elaborates IAE, adding "syntactic sugar" which serves to identify additional important computational structures expressed by IAE. The resulting language is called Iswim (If you See What I Mean), which Landin presents as a framework providing mechanisms for naming things (binding) and for defining functional relations. The point is to reduce language design to a matter of choosing primitive data and operations and choosing printed and physical representations.

The fact that in AE, the value of an application $e_0(e_1)$ depends only on the values of the component expressions and not on the computations described by the component expressions suggests that there is an equivalence on expressions satisfying laws analogous to those of the lambda calculus (and possibly additional laws). Landin gives some axioms for equivalence of Iswim expressions. Some of the axioms are "sugar elimination" axioms that serve to define the added syntactic entities of Iswim in terms of IAE expressions. The more substantial axioms fall into three categories: substitutivity, naming, and simplifications. Substitutivity has to do with determining when a subexpression can be replaced by an equivalent one without changing the equivalence class of the whole expression. For example

$$e_0 \equiv e_1 \rightarrow e(e_0) \equiv e(e_1).$$

Naming has to do with what instances of beta conversion are allowed. In general such conversions are restricted to AEs since they affect the order of evaluation of expressions. Simplifications are equivalences related to primitive operations. For example, axioms for properties of pairing and projection operations would fall in this category as would axioms about distribution of application over a conditional operator. No axioms for formation of abstraction are given. Neither are any criteria for inequivalence given.

Remarks

The initial motivation given by Landin for developing the theory of applicative expressions is analogous to (and derives from) Church's motivation for developing the lambda notation and the lambda calculus as stated in the introduction of Church [1941]: "to analyze the concept of function as it appears in various branches of mathematics". Landin was working with languages for describing computations to computers while Church was interested in understanding properties of functions independent of their use in a particular context.

AE is a simple, elegant and powerful language. Landin presents expressions and other objects as abstract data structures, using abstract syntax, and formulates computation rules independently of the given data and primitive operations. The function and argument positions in application expressions are treated symmetrically. The SECD abstract machine gives a simple interpretation of expressions as rules for computation.

In McCarthy [1963b] a similar separation of data and computation rules is made. This formalism is limited to first-order functions and does not treat function or control abstractions. Systems of recursive definitions can be viewed as a fragment of AE by assuming conditional to be among the primitive operations.

No dialect of Lisp provides meaning of programs independent of the S-expression world; S-expressions are a fundamental part of Lisp. The main Lisp dialects evaluate the function position according to special rules – partly because Lisp was originally conceived as as first-order language with lambda as a notational convenience and partly for reasons of efficiency.

Landin's environments and closures are data abstractions of the a-lists and funargs of Lisp 1.5. Program points are more general and mathematically simpler than the labeling and jumping mechanisms of Algol60 or catch and throw of Maclisp. Environment descriptors of Interlisp provide capability of implementing both closures and program points and Scheme continuations are equivalent to program closures.

I.4.2. Information structures for modeling computation

Wegner [1971] proposed the use of *information structures* for modeling computation. The idea is to characterize computations by the class of data structures generated during computation and by the transformations acting on these structures. An information structure is a triple $<I, I_0, F>$ where I is a set of data structures, I_0 is the subset of I consisting of the initial state structures, and F is a set of transformations on I.

Automata, digital computers, and programming languages can be modeled uniformly using information structures. By modeling computation using information structures one can reduce proving properties of programs to proving properties of the underlying structures, or of equivalence classes of structures. This paradigm also provides a framework for formal treatment of compiling since an important aspect of a compiler correctness proof is proving equivalence of source and target language interpreters.

Among the examples treated by Wegner are Algol60-like languages, the Burroughs B6500 computer, and Snobol. He does not attempt to treat abstraction and application as computation primitives.

An important example of the information structure approach is the Vienna Definition Language (VDL) (Wegner [1972]). This is a programming language for defining programming languages. The information structures are tree structures. The leaves are called elementary objects and composite trees are finite sets of selector tree pairs. Thus components of a composite tree are unordered and a subtree in given by path which is a sequence of selectors.

Programs are represented as trees – corresponding directly to the use of abstract syntax to specify the syntax of a programming language (McCarthy [1963a]). The semantics of a programming language is defined in terms of sequences of transformations on computation states. The key component of a computation state is the control tree. Transformations fall into two categories (i) macro instructions and (ii) value returning instructions. A macro instruction causes a leaf of the control tree to be replaced by a subtree of instructions. No other component of the computation state is affected. A value returning instruction deletes a leaf of the control tree having an immediate value and propagates the value to predecessor nodes. It may also affect other components of the computation state.

An important application of this approach to semantics is the development of methods for proving equivalence of different representations of computation state and rules for evaluation for a given language. An example is given proving equivalence of two semantics for a simple block structured language. VDL is also defined in VDL.

Remarks

Landin's SECD machine can easily be formulated in the information structure framework. The important feature is that the SECD structures are highlevel data abstractions and naturally derived from the structure of expressions. The Interlisp environment descriptors can also be thought of as information structures.

The information structure approach seems ideal for treating intensional properties of programs, but it does not seem to have been used for this purpose.

I.4.3. Call-by-value lambda calculus

Since applicative expressions are just terms of the lambda-calculus, the question arises as to how the notions of value and evaluation compare to the notions of reduction and normal form. In order to make meaningful comparisons it is necessary to distinguish between two basic choices for evaluation rules. According to the transition rules for the SECD machine, in order to evaluate $e_0(e_1)$ the value of e_1 is computed, then the value of e_0 is computed, then application is carried out (if meaningful). The value of the argument must be computed even if it is not used in computing the value of the application. Thus we say SECD machine evaluation obeys the *call-by-value* evaluation rule. An alternative is to first evaluate e_0 and, in the case that the value is a closure, substitute the argument expression for the bound variable rather than evaluating it. This is a *call-by-name* evaluation rule. Note that the function given by a lambda-expression using the call-by-name rule may be defined for some arguments for which computation would not terminate using the call-by-value rule. The classical example (derived from Morris [1968]) is the function defined recursively by

$$f(x,y) \leftarrow \mathbf{if}(x = 0, 1, f(x-1, f(x,y))).$$

If evaluated call-by-value $f(0) = 1$ and for $n > 0$ the computation of $f(n)$ does not terminate. If evaluated call-by-name $f(n) = 1$ for any natural number n. The order and number of times that a subexpression is evaluated also depends on the choice of evaluation rule. Thus in a language, such as IAE, with escape mechanisms or in which evaluation of expressions may have "side-effects", the value returned will in general depend on the choice of evaluation rule.

We will focus on the results for the case of call-by-value evaluation. Plotkin [1975] defines a call-by-value variant of the lambda calculus, Lambda-v. The language of Lambda-v is a variant of AE using constant symbols rather than environments. There is a constant symbol for each element of the given data domain and for each primitive function. Thus for every expression-environment pair of AE there is a corresponding closed term in the extended language obtained by hereditarily substituting constant or operation symbols for variables bound to data or primitive operations, and substituting closed lambda terms for variables bound to closures. For any closed expression of the extended language, there is an expression-environment pair of AE that corresponds to this expression. If two expression-environment pairs of AE correspond to the same expression then they have corresponding values (or both are undefined). Thus for closed e we say e evaluates to v if SECD computation of some corresponding expression-environment pair returns a value corresponding to v. We say e is undefined if SECD computation of some corresponding expression-environment pair does not return a value.

A *value expression* is a variable, a constant, or an expression of the form $\lambda(x)e$. For simplicity, primitive functions are restricted to act only on data but may return any closed value expression. The reduction rules for Lambda-v are the usual rules for the lambda calculus (see Church [1941] or Barendregt [1981]) with two exceptions: (i) the β rule which is replaced by the β-v rule and (ii) a δ-rule added for primitive application.

(β-v) $\{\lambda(x)e\}e_1 \geq_v e|^{x}_{e_1}$ if e_1 is a value expression

(δ) $f(d) \geq_v ap(f,d)$ if f is a primitive function symbol,
d is a data symbol and $ap(f,d)$ is defined

$\equiv_v$ is the symmetric closure of $\geq_v$. An expression is in value normal form (VNF) if it has no subexpression to which the (β-v) or (δ) rule applies.

The key theorems for Lambda-v and values are

(Substitutivity) If $e_0 \equiv_v e_1$ and e is a value expression then $e_0|_e^x \equiv_v e_1|_e^x$

(Church-Rosser property) If $e \geq_v e_0$ and $e \geq_v e_1$ then there is e_2 such that $e_0 \geq_v e_2$ and $e_1 \geq_v e_2$

(Evaluation) For closed e

(i) if e evaluates to v then $e \geq_v v$

(ii) if $e \equiv_v v'$ for some value v' then e evaluates to v for some value v

Some additional points of interest are

(Non-substitutivity) The value requirement for (Substitutivity) is necessary. A counter example is $e_0 = \{\lambda(x)\lambda(x)x\}x$, $e_1 = \lambda(x)x$, and $e = \{\lambda(x)x(x)\}\lambda(x)x(x)$.

(Non-Church-Rosser) If value is replaced by VNF in the β-v rule, the rules no longer have the Church-Rosser property.

In addition to Lambda-v equivalence, Plotkin defines a natural notion of equivalence of expressions called *operational equivalence.* Intuitively, two expressions are operationally equivalent iff one can replace the other in any closeed term without changing the "meaning" of the term. More precisely, the operational equivalence relation $\approx$ is defined by $e_0 \approx e_1$ iff for any expression context $C\{\ldots\}$ such that $C\{e_0\}$ and $C\{e_1\}$ are both closed, either (U) holds or (V) holds.

(U) Both $C\{e_0\}$ and $C\{e_1\}$ are undefined (SECD computation does not return a value).

(V) There are closed values v_0, v_1 such that $C\{e_0\}$ has value v_0 and $C\{e_1\}$ has value v_1 and if either v_0 or v_1 is a data constant then $v_0 = v_1$.

The relation between operational equivalence and Lambda-v equivalence is

(i) $e_0 \equiv_v e_1$ implies $e_0 \approx e_1$

(ii) The converse of (i) does not hold. $\lambda(x)x(\lambda(y)x(y))$ is operationally equivalent to $\lambda(x)x(x)$ but $\lambda(x)x(\lambda(y)x(y)) \equiv_v \lambda(x)x(x)$ can not be derived using the lambda rules.

I.4.4. Meta-programming

Morris [1970] (see also Reynolds [1972]) proposed using simple applicative languages such as AE, which have comparatively well-understood semantics, to define the semantics of more complex languages. This approach makes formal language definition a (meta) programming activity – namely programming functions that compute the meaning of program expressions. Morris outlined a number of ideas about how this could be accomplished. For example, assuming that the given primitive operations for AE include conditional and that the given data include data structures for expressions, closures, environments, and dumps the evaluation function determined by the SECD transition rules can be directly described by an AE program. Alternatively, using application and abstraction, objects of the semantic domain such as closures, environments and program closures can be represented as AE functions rather than as data structures. To describe non-lexical control sequencing constructs such as goto, escape mechanisms such as error exits or catch and throw, or co-routine mechanisms, it is necessary to specify the continuation of a computation at each point of a program. To define semantics of languages with updating or non-lexical control, it is also necessary to insure that evaluation is carried out in the proper order - to maintain control over control.

Reynolds [1972] studies *definitional interpreters* (programs defining semantic meaning functions), taking both the defining and defined languages to be variants of AE. An important question to consider is the affect of the semantics of the defining language on the semantics implied for the defined language. For example is the evaluation rule obeyed by the defined language affected by the choice of evaluation rule for the defining language? Can the defined language be extended with out major modification of the interpreter? The choice of control structure for the interpreter is the key for these questions. Another question is what information is gained about data structures needed to carry out computations. The choice of semantic domains is important here.

Reynolds defines four interpreters, combining different choices of control structure and semantic domain. The choices for control structure are meta-circular (Mc) (terminology due to Reynolds) and continuation-passing (Cp). A meta-circular interpreter interprets constructs of the defined language by using the corresponding construct of the defining language. This is analogous to the informal use of "and" and "for all" to define the semantics of the logical connectives $\wedge$ and $\forall$. The computation context built up in carrying out the defined computation is represented by that built up in carrying out the defining computation. A continuation-passing interpreter represents the context built up in carrying out the defined computation as an explicit parameter of the interpreter (the continuation). No context is built up in the defining computation. The choices for representing objects of the semantic domain are (ho) as functions defined by AEs – making the interpreter higher-order, and (fo) as abstract data structures – making interpreter first-order.

The four interpreters are related by informal transformations illustrating the passage from meta-circular to continuation-passing, passage from higher-order to first-order, and passage from first-order to higher-order.

$$(\mathrm{Mc.ho}) \mapsto (\mathrm{Mc.fo}) \mapsto (\mathrm{Cp.fo}) \mapsto (\mathrm{Cp.ho})$$

This provides useful insight into the relations and distinctions between the various choices.

To illustrate the issues discussed above we look in more detail at some of the features of these interpreters. The application and lambda clauses for the higher-order meta-circular interpreter (Mc.ho) satisfy

$$\begin{aligned} eval(\llbracket e_0(e_1)\rrbracket, \xi) &\sim \{eval(\llbracket e_0\rrbracket, \xi)\}eval(\llbracket e_1\rrbracket, \xi) \\ eval(\llbracket \lambda(x)e\rrbracket, \xi) &\sim \lambda(v)eval(\llbracket e\rrbracket, \xi\{\llbracket x\rrbracket \leftarrow v\}) \end{aligned}$$

ξ is an environment - here a function mapping (codes for) variables to values and $\xi\{\llbracket x\rrbracket \leftarrow v\}$ is the function given by $\lambda(a)\mathbf{if}(a = \llbracket x\rrbracket, v, \xi(a))$

A meta-circular interpreter is concise and elegant, but uninformative in that it presumes the control constructs are already understood.[1] Since for a meta-circular interpreter the meaning of application in the defined language is directly determined by that of the defined language, the rule obeyed by the defining language is inherited by the defined language.

The interpreter (Cp.ho) is derived from (Mc.ho) by a transformation to continuation-passing style. The transformation introduces an additional parameter which represents the remainder of the computation at each stage and unwinds complex applications so that computation proceeds by construction of continuations and applying them to results of basic computations guaranteed to terminate. A continuation-passing interpreter provides a definition that is independent of the evaluation rule obeyed by the defining language. The key observation is that the value of an expression only depends on the evaluation rule if, in the course of evaluation, an application expression is encountered in which evaluation of the argument causes a function call that does not terminate.

Letting γ stand for a continuation, the application and lambda-clauses for the (Cp.ho) satisfy

$$\begin{aligned} eval(\llbracket e_0(e_1)\rrbracket, \xi, \gamma) &\sim eval(\llbracket e_0\rrbracket, \xi, \lambda(f)eval(\llbracket e_1\rrbracket, \xi, \lambda(v)f(v, \gamma))) \\ eval(\llbracket \lambda(x)e\rrbracket, \xi, \gamma) &\sim \gamma(\lambda(v, \gamma_0)eval(\llbracket e\rrbracket, \xi\{\llbracket x\rrbracket \leftarrow v\}, \gamma_0)) \end{aligned}$$

By replacing the higher-order objects in the semantic domain such as environments, closures, and continuations by data structures, the higher-order interpreters can be transformed into first-order interpreters. One distinction between first-order and higher-order is that higher-order avoids (ugly) details about the

[1] The existence of a meta-circular interpreter is a kind of closure condition on the computation primitives and underlying data domain. It means that the data structure is adequate to encode the syntax and the control structure is adequate to describe the computations without further encoding of information as data.

data structures (defining constructors, selectors, ...). Of course, defining a first-order interpreter gives more information about the data structures generated in the process of carrying out a computation – precisely those ugly details about constructors, selectors, ... – and provides a basis for representing intensional properties.

(Cp.fo), the first-order continuation-passing interpreter, naturally determines a machine analogous to the SECD machine. Cp-machine states are the argument triples $<[\![e]\!], \xi, \gamma>$ of (Cp.fo). The clauses in the definition of (Cp.fo) correspond directly to transition rules. Thus a (Cp.fo) style interpreter is very close to the information structure approach of modeling computation.

A continuation-passing interpreter is readily extended to imperative constructs such as assignment and escape or labeling mechanisms. This is illustrated by adding expressions **escape**$(x)e$ to the defined language. Escape expressions are the natural analog of Landin's J for the Cp machine. To evaluate **escape**$(x)e$ the escape function corresponding to the current continuation is bound to x and evaluation of e is carried out in the extended environment. When the escape function corresponding to a continuation γ is applied to a value and another continuation γ_0, computation proceeds by returning the value to γ. Thus escaping from the context γ_0 to γ. The (Cp.ho) clause for evaluating an escape expression satisfies

$$eval([\![\mathbf{escape}(x)e]\!], \xi, \gamma) \sim eval([\![e]\!], \xi\{[\![x]\!] \leftarrow \lambda(v, \gamma_0)\gamma(v)\}, \gamma)$$

In order to obtain meta-circular a definition of the language with **escape** added, it is necessary to extend the defining language as well.

I.4.5. Scott-Strachey semantics

"Scott-Strachey" semantics is a method for defining semantic meaning functions for arbitrary languages based on giving domain equations for syntactic and semantic domains and giving semantic equations, expressed in an AE-like language, for meaning functions. An important goal guiding the choice of semantic domains is that semantic meaning functions should be *compositional.* That is, the meaning of an expression should be determined by the meanings of its sub-expressions, and should not depend on the context in which it is contained. Wadsworth independently developed similar ideas to those of Morris for handling non-local jumps using continuations and these ideas were applied by Strachey and Wadsworth [1974] to extend the basic Scott-Strachey methods.

Typically, in the Scott-Strachey approach, the semantics of the defining language is given by interpreting lambda expressions in extensional models (Scott models) constructed by solving domain equations using structures that are complete partial orders. The solutions to the semantic equations are functions in the extensional sense of being determined by their graph. Higher order interpreters such as (Mc.ho) or (Cp.ho) will thus assign extensional denotations to higher order objects of the defined language. First order interpreters such as (Mc.fo) or (Cp.fo) will however assign "codes" as denotations to higher order objects of the defined language. By suitable choice of semantic domains, the Scott-Strachey methods could be used to interpret programs by transitions on information structures in the sense of Wegner, although we are not aware that this has in fact been done.

In contrast, if the defining language AE is interpreted by the SECD machine, the functions given by lambda expressions are functions in the intensional sense of being rules for computation. One can infer equivalence relations for defined expressions from lambda-v equivalence or operational equivalence relations for AE.

Scott models will be discussed further in §5. The basics of the Scott-Strachey approach can be found in Milne and Strachey [1976]. For a presentation of domain theory (solving domain equations in complete partial orders) we recommend notes of Plotkin [1978].

I.4.6. More about continuation-passing

The transformation to continuation-passing style can be carried out quite generally. Using the formulation of AE in terms of purely syntactic expressions plus environments for interpreting symbols, such transformations can be defined in a manner independent of the choice of basic data and primitive operations. A continuation-passing fragment of AE is the image of such a transformation. It has a number of interesting properties –

(i) the argument component of an application is a primitive expression – a symbol or a primitive function applied to a list of symbols

(ii) no intermediate values are returned – computation proceeds by constructing continuations and applying continuations to intermediate results obtained by evaluation of primitive expressions

(iii) no external state is built up during computation – all information about the computation state is represented explicitly

We have seen how (i) and (iii) are used to insure that interpreters give definitions independent of the evaluation rule obeyed by the defining language and for interpreting control abstractions such as program points or escape expressions.

Fischer [1972] used property (ii) to show that any applicative expression can be transformed to an expression that defines the same function on the data domain, but which forces the dynamic scope of variable binding to extend over the dynamic lifetime of any closures generated. Thus a program using functions as values only as intermediates can be transformed to one which will behave correctly in systems such as Maclisp where funarg objects can not be passed out of their dynamic scope. Fischer proves the following relation between programs and their transformations.

(iv) If p is a program that defines a function on data, p' is the transformation of p to continuation-passing style, and Id is the identity continuation then for any continuation c and any element d of the data domain

$$p'(d,c) \sim c(p(d)) \qquad \text{hence} \qquad p'(d,Id) \sim p(d).$$

The Rabbit compiler for Scheme (Steele [1978]) uses a transformation to continuation-passing style as its core. The point here is that the continuation-passing fragment can be viewed as an abstract machine represented in the original language. The advantage is that the same tools for analysis and optimization can be used at both ends of compiling process, and compiled code can be directly interpreted with no added fuss.

I.5. Applicative structures

I.5.1. Recursion theory

Recursion theory has its roots in the work of Peano and Dedekind and their studies of the natural numbers and of the definition of functions by induction on numbers, i.e. by primitive recursion. The goal of early work in recursion theory was to characterize the class of functions, say on natural numbers, computable by algorithms. This gave rise to a number of equivalent notions including lambda-definability (Kleene [1936a], equation calculi (Gōdel [1934], Kleene [1936b]), and Turing machines (Turing [1936]). Later work in recursion theory has focused attention on the presentations and on the structure of classes of functions. Some of the means of presenting such classes are

(i) equation calculi – using expressions and rules for deducing equations from systems of defining equations (see Kleene [1952], McCarthy [1963b])

(ii) schemata (comprehension principles) for introducing functions – the schemata serve as rules for generating a class of functions from a given set of basic functions (see Kleene [1952] and [1959])

(iii) inductive definability – a class of function(als) given as fixed points of a system of monotone functionals (see Platek [1966], Kechris and Moschovakis [1977], Feferman [1977], Moschovakis [1984])

(iv) invariant definability – for example functions given as solutions to systems of equations (not the same as (i), which also involves rules for deduction, see Grzegorczyk, Mostowski and Ryll-Nardzewski [1959])

Of particular interest in the study of the structure of classes of functions was the study of hierarchies of functions and predicates over the natural numbers. Additional notions include relative recursion – one function being computable from another (an oracle), degrees – equivalence classes of functions each computable from the other, and recursion at higher types. From this work came tools for studying the fine structure of mathematics and for measuring the strength of formal systems according to the complexity of functions that can be defined.

The scope of recursion theory has been extended to consider recursion over structures other than the natural numbers – for example recursion on ordinals and recursion over arbitrary abstract structures. Principal objectives of this work are to look for abstractions and axiomatic characterizations of fundamental features of the class of partial recursive functions on natural numbers and to find meaningful abstract versions of Church's thesis.

To present the main ideas and results from recursion theory relevant to $\mathcal{R}um$, we introduce informally notions of *a recursion theory* and *a computation theory.* A recursion theory on a domain C is a class F of partial functions and functionals on C satisfying certain *closure* conditions such as closure under explicit definition, definition by cases, fixing of parameters (S^m_n theorem), and recursion. The following are versions of the S^m_n and recursion theorems, which are the key theorems for our purpose. (The subscript Rt is to distinguish these theorems from the corresponding theorems for a computation theory which are labeled by the subscript Ct.)

- (S^m_n theorem)$_{Rt}$ If ϕ is an $m{+}n$-ary function in F and $c_1, \ldots, c_m$ are elements of C then $\lambda(y_1, \ldots, y_n)\phi(c_1, \ldots, c_m, y_1, \ldots, y_n)$ is an n-ary function in F.
- (Recursion theorem)$_{Rt}$ If $\Phi(f, x_1, \ldots, x_n)$ is a functional in F with n-ary function parameter f and Φ is monotone in f then the least fixed point Φ^∞ of Φ with respect to f is in F. Φ^∞ satisfies

$$\Phi^\infty = \Phi(\Phi^\infty) \quad \text{and} \quad \phi = \Phi(\phi) \to \Phi^\infty \sqsubseteq \phi.$$

$\sqsubseteq$ denotes the usual partial ordering on partial functions, i.e. the subset ordering the graphs. In general the functional Φ may have function and individual parameters not shown above. These parameters will also be parameters of the least fixed point Φ^∞.

The partial recursive functions and functionals on natural numbers (see Kleene [1952], chapter XXII) are perhaps the simplest example of a recursion theory. The notion of a ***suitable class*** of functionals defined by Kechris and Moschovakis [1977] or that of a ***precomputation theory*** (see Fenstad [1980], Definition 1.1.10) are typical examples of closure conditions for recursion theories. Recursion theories are often defined as the least class of functionals containing some base set and closed under certain operations. The $\mathcal{I}$- recursive functionals for suitable $\mathcal{I}$ defined in Kechris and Moschovakis [1977] and the theories of hereditarily consistent functionals given by Platek [1966] are examples of such recursion theories. An example of $\mathcal{I}$- recursive functionals of particular interest for computing is given in Moschovakis [1984] (see §VIII.3).

A computation theory over a domain C is a set F of codes describing computations on C together with a ternary application relation $\{f\}(x_1,\ldots,x_n) \sim z$ on $F \times C^* \times C$, and a well-founded partial ordering $\prec$ on tuples of the application relation. C^* is the set of finite sequence from C and $\{f\}(x_1,\ldots,x_n) \sim z$ means that the computation coded by f applied to the sequence $x_1,\ldots,x_n$ returns z.[1] The ordering provides information about the structure of the described computations and is typically induced by a rank function that measures the size of the computations by assigning ordinals to tuples of the application relation. The class of codes for a computation theory must also satisfy certain ***closure*** conditions analogous to those for a recursion theory. The following are versions of the S^m_n and recursion theorems for computation theories.

- $(S^m_n \text{ theorem})_{Ct}$ For each m, n, there is a computable function S^m_n such that for each $f \in F$ and $x_1,\ldots,x_m,y_1,\ldots,y_n \in C^*$:

$$S^m_n(f,x_1,\ldots,x_m) \in F,$$

$$\{S^m_n(f,x_1,\ldots,x_m)\}(y_1,\ldots,y_n) \sim z \leftrightarrow \{f\}(x_1,\ldots,x_m,y_1,\ldots,y_n) \sim z.$$

and

$$(\{f\}(x_1,\ldots,x_m,y_1,\ldots,y_n) \sim z) \prec (\{S^m_n(f,x_1,\ldots,x_m)\}(y_1,\ldots,y_n) \sim z)$$

- $(\text{Recursion theorem})_{Ct}$ There is a computable function R such that for $f \in F$ and $x_1,\ldots,x_n \in C^*$

$$R(f) \in F,$$

$$\{R(f)\}(x_1,\ldots,x_n) \sim z \leftrightarrow \{f\}(R(f),x_1,\ldots,x_n) \sim z$$

and

$$(\{f\}(R(f),x_1,\ldots,x_n) \sim z) \prec (\{R(f)\}(x_1,\ldots,x_n) \sim z)$$

[1] Here we are using braces to denote the function computed by a code following Kleene. We will call these "Kleene braces". This is in contrast to our usual convention of using braces simply for grouping. Context will make it make it clear which usage is intended.

By a suitable encoding of the presentation, many recursion theories are readily transformed into computation theories. For example, the partial recursive functions and functionals on natural numbers become a computation theory by assigning Gödel numbers to schemata describing functions and using the natural partial ordering implicit in the inductive definition of the function described by the schemata. Prime and search computability over an abstract structure (Moschovakis [1969], see also below) are examples of computations theories.

A further important theorem in recursion theory is the normal form theorem. This theorem says that each computable function can be computed by iterating some function from an "elementary" class of functions (with additional elementary operations allowed at the beginning and end of the computation.) One version of this theorem is based on the Kleene T-predicate for partial recursive number functions (Kleene [1952]).

- (Normal Form theorem) There is a primitive recursive function U and for each n a primitive recursive predicate T_n such that for each n-ary partial recursive function ϕ there is number e such that

$$\phi(x_1,\ldots,x_n) \sim z \leftrightarrow U(\mu(y)T_n(e,x_1,\ldots,x_n,y)) \sim z$$

Here μ is the least number operator – for any predicate Ψ, $\mu(y)\Psi(y)$ is least y such that $\Psi(y)$ and $\mu(y)\Psi(y)$ is undefined if no such y exists. The proof of the normal form theorem is based on the notion of stepwise generation of a computation structure which when complete gives the value computed. For example $T_n(e,x_1,\ldots,x_n,y)$ says that y codes a completed computation of $\{e\}(x_1,\ldots,x_n) \sim z$ for some z. For such y, U extracts the value, $U(y) = z$. The normal form theorem is related to the conversion of recursion to iteration-plus-stack (y playing the role of stack) and hence to the process of compiling. It says that "machine computations" are represented by a fragment of the given theory in such a way that all computations of the theory are represented in that fragment.

An important application of recursion theory is the notion of realizability (see Kleene [1952], §82). This notion was introduced in order to prove a conjecture that if a closed formula of arithmetic $(\forall x)(\exists y)A(x,y)$ is intuitionistically provable, say in Heyting arithmetic, then there is a general recursive function ϕ such that $(\forall x)A(x,\phi(y))$. The proof of the conjecture shows how to obtain a numeric code for ϕ from a proof of $(\forall x)(\exists y)A(x,y)$. The code is called a realization of the formula $(\forall x)(\exists y)A(x,y)$. This was the first of many methods developed for extracting 'programs' from proofs.

Remarks.

- Many of the key ideas in basic recursion theory are due to Kleene. These include the above theorems, the generalization from total to partial functions, representation of the process of computation as one of generating computation trees, the idea of coding computation structures as data, and the notion of realizability.

- Our formulation of a computation theory is based on that of Moschovakis [1971] (see also Fenstad [1980]). The notion of a recursion theory is a "code-free" or extensional version of this notion. A computation theory also determines a class of computable functions which constitute a recursion theory.

- The S^m_n theorem for a computation theory captures the essential aspect of returning functions as values. It corresponds closely to closure formation à la Landin (§4). One can think of closure formation as a direct implementation of the S^m_n theorem.

- The recursion theorem stated for a recursive function theory is essentially Kleene's first recursion theorem (Kleene [1952] Theorem XXVI) while the version stated for a computation theory corresponds to Kleene's second recursion theorem (Kleene [1952] Theorem XXVII). The distinction is that first recursion theorem concerns functions as fixed points of computable functionals while the second recursion theorem concerns codes describing functions and solutions to equations expressed in terms of the application relation. In general "minimality" is not a meaningful notion for codes. Our version of the second recursion theorem is somewhat stronger than is usually stated, although in practice it is often the version that is proved.

For references to original papers and a history of the development of the main ideas of recursion theory see Kleene [1979]. For further reading we recommend Kleene [1952], Fenstad [1980], and the section on recursion theory in Barwise [1977].

I.5.2. Recursion on abstract structures

Moschovakis [1969] develops a recursion theory constructed uniformly over a given structure $\langle B, \bar{\phi} \rangle$ with given domain B and given functions $\bar{\phi}$. The computation domain B^* is obtained from the B by adding a new element (called 0) and closing under pairing.[2] 0^* denotes the subset of B^* generated from 0. Numbers are

[2] The * in B^*, following the notation of Moschovakis, is not to be confused with our normal use as in C^* to denote sequences from C.

represented as elements of 0^*. A computable function is assigned to each element of B^* by an inductive definition of the application relation

$$\{e\}(x_1, \ldots, x_n) \sim z.$$

As in our description of a computation theory, $\{e\}(x_1, \ldots, x_n) \sim z$ means that the value of the function coded by e at the sequence $x_1, \ldots, x_n$ is z. There are seven rules that generate the functions on B^* primitive recursive in $\bar{\phi}$. The remaining (8th) rule is that of reflection

$$\{e\}(x_1, \ldots, x_n) \sim z \;\mapsto\; \{< 8, n+m+1, n >\}(e, x_1, \ldots, x_n, y_1, \ldots, y_m) \sim z$$

which provides for the interpretation of an argument as a code and is the key to describing general recursions. The class of functions computed in this theory are called the prime computable functions over the given structure $\langle B, \bar{\phi} \rangle$.

Closure under composition, substitution, and projections and the S^m_n and recursion theorems are proved by giving codes for the corresponding functionals as functions of codes for the parameter functions. Although arbitrary elements of B^* may appear in codes the 'control' information is coded using only elements of 0^* and equality of the control parts of codes is computable.

The presentation of this theory provides not only a recursion theory, but also a computation theory. The inductive definition of the application relation naturally determines a set of codes that describe computations. In addition a subcomputation relation, a tree structure for computations and a process for carrying out computations can be derived from the inductive definition of the application relation.

The Moschovakis theory as presented, is not adequate for a theory of symbolic computation. The principal difficulties are the choice of basic computation primitives, and the ability (in fact necessity) to use operations on codes to describe computations. More precisely

- the representation of codes as "Gõdel" numbers in 0^* introduces irrelevant detail. It is the operations constructing codes that are important for defining the application relation and for defining operations on codes.
- computation is described in terms of codes for functions rather than being given in terms of expressions and environments.
- the choice of codes and computation primitives does not allow extensional (code independent) representation of function abstraction or of control structures such as combinators, iterators, loops, etc.
- this theory cannot admit general meaning preserving operations since in particular equality of the control parts of codes is computable.

To illustrate the final point, suppose c is a code and $'$ is a transformation such that c' computes the same function as c. Suppose further that

$$\{f\}(e,x) \sim \begin{cases} 0 & \text{if } e = c \\ 1 & \text{if } e \neq c \end{cases} \quad \text{and} \quad \{f'\}(e,x) \sim \begin{cases} 0 & \text{if } e = c' \\ 1 & \text{if } e \neq c'. \end{cases}$$

Then the function computed by f is not preserved by the transformation $'$.

Remarks

- The class of functions defined by systems of recursion equations over a given data structure (McCarthy [1963b], see §2) and AE (Landin [1964], §4) are also recursion theories over abstract structures. The recursion theoretic aspects of these formulations were not developed as the interest was in applications to programming, programming language semantics and proving properties of programs.

- Moschovakis [1984] (see §VIII.3) presents a recursion theory over abstract structures in which both recursion theoretic and computation theoretic aspects are developed.

I.5.3. Extensional models of the lambda calculus

There are now well-developed methods of constructing mathematical models of the lambda calculus. The basic ideas derive from work of Scott. (See Barendregt [1981] chapter 18 for details and references). The general construction can be expressed quite simply in terms of complete partial orderings (c.p.o.s). Fix a c.p.o. $<D, \sqsubseteq>$ with domain D and partial ordering $\subseteq$. Then D has a unique least element $\perp$ (bottom) and if X is a non-empty directed subset of D (every pair of elements of X have an upper bound in X) then X has a least upper bound $\sqcup X$ in D. We read $d_0 \sqsubseteq d_1$ as d_0 is less defined than d_1. There is a natural topology on D derived from the partial ordering such that continuous functions are those which preserve least upper bounds of directed subsets – $f(\sqcup X) = \sqcup f(X)$. Let D' be the space of continuous functions on D. A key fact is that each f in D' has a least fixed point **fix**(f) in D. (**fix** is in fact continuous.) To define a model of the lambda calculus it is sufficient to define a continuous function **fun** from D to D' and a continuous function **graph** from D' to D such that **fun**(**graph**(f)) $= f$. **fun** interprets element of D as codes of continuous functions and **graph** assigns to each continuous function a code in D that computes it. Thus there is a natural binary application operation on D given by $d_0 \circ d_1 = \{\mathbf{fun}(d_0)\}(d_1)$. Using the **graph** and **fun** functions the applicative structure $<D, \circ>$ can be considered as a model of the lambda calculus. If it is also the case that **graph**(**fun**(d)) $= d$ then the model is extensional – two elements code the same function iff they are identical.

Two important examples are the D_∞ model (due to Scott) and the graph model $P\omega$ (due independently to Plotkin and Scott). For the graph model D is the power set of the set of natural numbers $P\omega$ partially ordered by subset inclusion. This model also satisfies $d \sqsubseteq \mathbf{graph}(\mathbf{fun}(d))$ and thus two elements in the image of **graph** code the same function only if they are equal.

The use of the graph model to provide semantics for AE-like languages is worked out in detail in Scott [1976]. A language LAMBDA is defined for describing computable functions on D.[3] Each LAMBDA expression e is given an interpretation as an element in D for each assignment of the free variables in e to elements of D. Many equations and theorems about LAMBDA computable functions are proved. In addition to the usual laws of the lambda calculus the laws of extensionality (ξ) and (ξ^*) hold in this model.

$$(\xi) \qquad (\forall x)(e_0 = e_1) \rightarrow \lambda(x)e_0 = \lambda(x)e_1$$

and

$$(\xi^*) \qquad (\forall x)(e_0 \sqsubseteq e_1) \rightarrow \lambda(x)e_0 \sqsubseteq \lambda(x)e_1.$$

The closure conditions for recursion theories are realized by LAMBDA computable functionals in a uniform manner. The recursion theorem in this model says that the combinator $\mathbf{Y}$ computes the least fixed point on graphs of continuous functionals.

$$(\mathrm{Y.fix}) \qquad f \in F \wedge d = \mathbf{graph}(f) \rightarrow \mathbf{Y}(d) = \mathbf{fix}(f)$$

Extensionality provides an important principle for proving LAMBDA equations and the recursion theorem provides an induction principle for proving properties of LAMBDA computable functions.

The graph model provides a rich equational theory for LAMBDA, however it does not provide an interpretation of expressions as descriptions of computation, and the only equivalence relation is the (fully extensional) equivalence of equality as functions.

[3] LAMBDA is AE with numbers, the arithmetic operations <+1, −1> and a conditional operation as the given primitives.

I.5.4. Non-extensional theories of partial operations

Feferman [1975] describes the construction of a theory $\tilde{T}$ of partial operations from a given theory T containing a minimal amount of coding power. Here "theory" means a language L, (constant, function, and relation symbols), a semantics (logical operations and their interpretation in a class of models - for example first-order logic), and a set of axioms. The goal of this work was to find "conservative extensions" of standard theories in which to formalize fragments of current mathematics, and in particular to provide natural means of defining the required functions and classes without introducing unneeded power. The key problems have to do with extensionality (abandoned by Feferman), and the ability to define function classes and power classes. The idea of the construction is to add a new ternary relation symbol $\sim$ to L (for partial function application) and use the coding power to represent function descriptions as codes. Axioms characterizing the application relation are added to the initial set of axioms. The unicity axiom (U) for $\sim$

$$\text{(U)} \qquad f(x) \sim y_0 \wedge f(x) \sim y_1 \rightarrow y_0 = y_1$$

expresses that the application relation is a partial function. A function comprehension principle (C_Φ) is added for each formula Φ monotonic in $\sim$.[4] (C_Φ) characterizes the function defined by Φ.

$$(\mathrm{C}_\Phi) \qquad \begin{aligned} &(\forall z_1,\ldots,z_n)((\exists f)(\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n) \sim f) \wedge \\ &\qquad (\forall x)(\exists! y)\Phi(x,y,z_1,\ldots,z_n) \rightarrow (\exists y) fx \sim y \wedge \\ &\qquad (\forall x,y)(fx \sim y \rightarrow \Phi(x,y,z_1,\ldots,z_n))) \end{aligned}$$

Here it is assumed that the free variables of Φ are among $\{x,y,z_1,\ldots,z_n\}$.

$$\iota(y)\Phi(x,y,z_1,\ldots,z_n)$$

is read "the y such that $\Phi(x,y,z_1,\ldots,z_n)$".

$$\lambda(z_1,\ldots,z_n)\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n)$$

denotes the closed term encoding the formula Φ, with the list $(x,y,z_1,\ldots,z_n)$ fixing an order for the free variables of Φ.

$$\lambda(z_{i+1},\ldots,z_n)\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n)$$

[4] Φ is monotonic in $\sim$ means that in any model, the set of elements satisfying Φ is a monotonic function of the interpretation of $\sim$.

abbreviates

$$\{\lambda(z_1,\ldots,z_n)\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n)\}(z_1,\ldots,z_i)$$

To prove that $\tilde{T}$ is a conservative extension of T, Feferman gives a method for constructing a model of $\tilde{T}$ from each model of T. Partial instantiations are interpreted by closure like objects. If f_Φ is the interpretation of the term

$$\lambda(z_1,\ldots,z_n)\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n)$$

and a_j interprets z_j then the tuple $<f_\Phi,a_1,\ldots,a_i>$ is the interpretation of

$$\lambda(z_{i+1},\ldots,z_n)\lambda(x)\iota(y)\Phi(x,y,z_1,\ldots,z_n).$$

Monotonicity of Φ allows successive approximations of $\sim$ to be constructed beginning with triples $<<f_\Phi,a_1,\ldots,a_n>,a,b>$ such that Φ has no occurrences of $\sim$ and $\Phi(a,b,a_1,\ldots,a_n)$ holds. The extended model is the limit of these approximations (iterated along the ordinals). The usual combinators k,s are definable and a recursion theorem is provable in the extended theory. Thus the extended theory naturally provides the functions lambda-definable from the given functions.

In contrast to Scott's models, there is for $\tilde{T}$ a non-extensionality theorem. Namely, it is inconsistent to assert that any two terms that code the same total function are equal.

(non.ext) $$\tilde{T} \vdash \neg(\forall f,g)(Tot(f) \wedge f \approx g \rightarrow f = g)$$

where $Tot(f)$ abbreviates $(\forall x)(\exists y)(fx \sim y)$, expressing that f codes a total function and $f \approx g$ abbreviates $(\forall x,y)(fx \sim y \leftrightarrow gx \sim y)$ expressing that f and g code the same function. Although extensionality and the notion of least fixed point are lost, the expressive power of classical logic (in particular negation) is maintained. Also, Feferman shows how to form further extensions providing a form of inductive definitions and notions of partial class, thus further enriching the theory for the purpose of formalizing parts of mathematics.

Leaving out the requirement of extensionality makes a conservative extension within the usual logical framework possible. However, no positive use of intensionality is made. Indeed, in ordinary mathematical practice, (the intended application) one is rarely if ever interested in the manner in which a function is computed, but rather in being able to prove properties relating argument-value pairs and in defining additional functions having given properties.

In Feferman's recursion theory it is possible to test for equality of codes. This is a key point in the proof of (non.ext) and for our purpose a principal difficulty with this formulation. The other key point in the proof is the uniform parameterization of descriptions - the direct implementation of the S^m_n theorem. The proof shows that these two features combined other minimal requirements are incompatible with extensionality. Note that the Moschovakis construction of a recursion theory over an abstract structure has the two essential features required for the non-extensionality theorem.

I.6. Towards a theory of symbolic computation

I.6.1. Summary

In this chapter we have examined a variety of aspects of symbolic computation, including the use of symbolic expressions to describe computation, the use of functional and control abstractions as programming tools, and the use of program transformations as a tool for constructing and reasoning about programs. In addition we discussed a variety of interpretations and applications of the language AE and key properties of structures modeling application and abstraction.

- Many of the programming examples can be directly expressed as AE programs and there is a rich literature on programming in languages like AE and its extensions.

- Program transformations were used informally to obtain definitional interpreters with alternative control structures and alternative semantic domains. Systems of formal transformation rules were used to derive programs with improved efficiency. Operations transforming intensional properties into extensional properties were used to analyze efficiency improving transformations and to represent other intensional properties of programs.

- Expressions of the language AE were interpreted as describing computations (Landin's SECD machine), as the language for an equational calculus (Plotkin's lambda-v calculus), and as defining functions (the graph model). Each of these interpretations provides a natural equivalence relation on expressions, addressing the general question of when it is meaningful to call two expressions the same. Operational equivalence (Plotkin) means being interchangeable in any program context without changing the operational meaning of the resulting program (as given by the SECD machine). Lambda-v equivalence means provably equivalent in the lambda-v calculus. Denotational equivalence means having the same denotation in the graph model.

- Two important fragments of AE were presented. One is the first-order fragment in which programs are given by systems of recursion equations and can be directly interpreted as partial recursive functions on the given data structure. The second is the continuation-passing fragment in which control is made explicit and which can be viewed as representing an abstract machine.

- The use of AE as a meta-programming language to define semantic meaning functions for general programming languages was also discussed. This includes Scott-Strachey semantics, extensions based on ideas of Morris and Wadsworth and Reynold's definitional interpreters.
- Recursion theories over abstract structures (Moschovakis) and theories of partial functions and classes (Feferman) illustrate methods for uniform constructions of applicative structures over given abstract structures or theories. The construction of recursion theories serves as a paradigm for construction of structures with computational content as well. The proof of the nonextensionality theorem for theories of partial functions and classes identifies problematic combinations of requirements for such structures – in particular the conflict between extensionality and the general ability to test for equality.

I.6.2. Symbolic computation systems

The reader may by now have accused us of "arguing from both sides of the mouth". On the one side we have said that two important features of Lisp are that programs are represented as data structures and that it is easy to write programs that operate on other programs. On the other side we have criticized recursion theories such as those of Moschovakis and Feferman for providing such capability.

The time has come to resolve this apparent inconsistency. In an underlying model of computation it is important to maintain the separation of the meaning of expressions from their representation as data and from the meaning given to the data operations. We call this *separation of control and data.* This separation is a form of abstraction that allows us to formulate and prove properties of programs and to define and study general operations on programs in a manner independent of the underlying data structure and of the representation of programs. Separation of control and data is also crucial for the purpose of defining meaningful equivalence relations on expressions and for defining transformations that preserve such equivalences. In contrast to a mathematical model of computation, a symbolic computation system such as Lisp is a dynamic, interactive system. It is essential for such a system to provide a mechanism for dynamic extension and tools for system building. This means being able to "discuss" the data structures used to represent computation and to "use" these structures within the same formalism. Thus programs and other components of the computation state must be represented as data.

In order to make clear the context in which the work on *Rum* is to be viewed and to show how we can have our cake and eat it too, we propose the notion of a *symbolic computation system.* The main components of such a system are:

- an underlying model of computation providing
  - a computation domain
  - symbolic descriptions of computation
  - structures for representing computations
  - rules for carrying out computations;
- an encoding of the model in a computation theory over a data structure (such as the S-expressions) so that
  - objects of the model are represented as data
  - operations and rules for computation are represented by computable functions; and
- reflection principles that provide for
  - conversion of computation state and descriptions into data
  - conversion of data into computation state and descriptions.

I.6.3. Goals

Rum is a step towards the goals for a mathematical theory of computation given by McCarthy [1963b]. The main goal of *Rum* is to provide an intensional, computationally meaningful interpretation of applicative expressions without prohibiting an extensional interpretation, and further to derive extensional interpretations from the basic computational interpretation. We want to represent properties of programs ranging from details of the computations described to properties of the functions computed; to represent naturally a variety of styles of programming; and to build on and extend existing work in logic, semantics, and program transformations. The main application is to express and prove properties of particular programs and classes of programs (functionals) and to study mathematical properties of computation mechanisms. The point is to provide tools both for programming and for the design of programming systems. These are rather different goals than those of work in logic establishing theories to account for mathematical practice. To make the distinction clearer the following summary compares and contrasts some of the general goals for work in foundations of mathematics and computing.

Goals for foundations of mathematics

- to account for practice – to provide precise definitions of informal concepts so that formal proofs can be carried out
- to isolate underlying principles for definition and proof and to determine what principles are needed for what parts of mathematics
- to isolate the proof theoretic strength of various fragments of mathematics

Goals for foundations of computing

- to account for practice – to provide precise definitions of informal concepts so that formal proofs can be carried out
- to improve practice – having a clear mathematical theory is important for making design decisions at all levels.
  - an understanding of the mathematical properties of computation mechanisms and of operations combining various mechanisms is a valuable tool for writing, debugging, and verifying programs.
  - an understanding of the mathematical consequences of combinations of computation mechanisms and of choices of computation structures and their representation is important for the design and implementation of programming systems.

I.6.4. About *Rum*

Rum provides a model of the applicative aspects of computation over generalized algebraic data structures. Thus it is (a fragment of) an underlying model of computation for a symbolic computation system. Implicit in the presentation of *Rum* is a natural encoding in the S-expression data structure. Addition of reflection principles will be discussed in §VIII.4.

I.6.4.1. Building on existing work

Rum is a further development of the work started by Landin [1964,1965,1966]. The basic approach is to view computation as a process of generating structures such as computation trees or sequences. This is a variation on the information structure approach of Wegner [1971]. The language for describing computation is AE extended by primitives for conditional and manipulation of sequences. Like AE, definitions are parameterized uniformly by a given data structure. The syntactic and semantic domains are abstract algebraic structures similar to those used by Reynolds [1972] for the first-order definitional interpreters. The difference is that computation rules are interpreted as rules for generating computation structures rather than as simply defining an evaluation relation. We go beyond existing work in developing mathematical properties of computations and proving properties of programs.

Rum is a "code-free" computation theory thus combining the extensional advantages of a recursion theory and the intensional advantages of a computation theory. In this light, *Rum* can be seen as a modification and elaboration of the construction of recursion theories over abstract structures given by Moschovakis [1969]. The main modifications are

- construction of B^* is replaced by construction of an abstract algebraic structure with purely syntactic entities (expressions à la AE). Values are constructed uniformly from the expressions and the data and operations of the given abstract structure.
- reflection is replaced by abstraction (closure formation) and application
- operations on codes are not provided as computation primitives - they are no longer needed
- the inductive definition of an application relation is replaced by a set of computation rules that serve as rules for generating computation trees and to present the definition of an evaluation relation defining the value of an expression in an environment

The induction principle and equations of Scott are obtained by defining an approximation relation on the computation domain, derived from the evaluation relation, which has the essential properties of $\sqsubseteq$.

I.6.4.2. The multitude of basic notions

The broad scope of *Rum* is essential for achieving the basic goals of a mathematical theory of computation. This entails developing a more complex theory than required for typical logical applications. In particular there are a variety of basic notions. (Even so *Rum* has a small number of primitives compared to practical programming systems!) Below we indicate briefly the main notions and the need for introducing them in accordance with the goals outlined above. Further remarks about our choice of basic notions will be found throughout the presentation and also in §VIII.2.

There are both syntactic and semantic domains in the *Rum* world. The semantic domains include both the computation domain and computation structures. The syntactic domain (*forms*) provides the ability to treat programs as data (objects to operate on) by providing primitive operations that characterize the structure of programs and from which general operations can be defined. Semantic domains are needed in order to assign meanings to programs. Having both a syntactic domain and a variety of semantic domains is needed in order to define operations on programs such as compiling, transforming, deriving, and constructing; to talk about the effects of these operations on the computations described; and to say what is changed and what is preserved.

The computation domain provides the basic objects used to assign operational and extensional meanings to programs. It contains data and operations from the given data structure, function abstractions (*pfns*), and control abstractions (*continuations*). Function and control abstractions are needed to represent naturally the variety of computation mechanisms that we wish to treat. Function and control abstractions are distinct from data. This provides the necessary separation of control and data that allows both intensional and extensional interpretations of programs. (Clearly, one does not want to prohibit the testing of equality between data items.)

There are three central notions for assigning intensional meaning to programs: descriptions, computation rules, and computation structures. Descriptions correspond to programs plus input. They are finite objects that contain information about how a particular computation is to be carried out. Computation is carried out by generating computation structures according to the computation rules. These structures provide the intensional interpretation of programs. Computation structures may be infinite, since computations may not terminate.

There are two basic computation structures – trees and sequences. Tree-structured computation provides for natural treatment of context independent computation mechanisms. Sequential computation provides for treatment of mechanisms such as escape and co-routining. Although sequential computations contain the tree-structured computations as a subset, it is important to develop the tree structured fragment separately. The latter has a much simpler and richer mathematical theory. Furthermore, context dependence in sequential computations is often localized and such computations can be treated essentially as tree-structured computations.

The computation rules are also interpreted as definitions of computation relations on descriptions. These relations provide a means of expressing extensional properties and of proving extensional properties by operational reasoning.

Chapter II. An illustrated introduction to Rum

This chapter is an informal introduction to *Rum*. The main concepts are illustrated by simple examples based on the problem of computing the product of numbers occurring at the leaves of a number tree. We begin with a recursive definition of the tree product function and then consider the problem of describing the computation in such a way that if a zero is encountered, it is returned as the value immediately without processing the remainder of the number tree. The examples illustrate the use of functional and control abstractions to describe computation; the process of generating computation trees and sequences; and properties of computations that can be read off completed computation structures or derived from the rules for generating computation structures. Equations satisfied by function and control abstractions are illustrated and we show how program transformation methods can be applied in situations where computations involve the generation and application of functional and control abstractions.

Rum notation will be used informally to present the examples with enough explanation, we hope, so that the reader can understand the particular cases, and also get some idea of the general concepts. Formal definitions will be given in the following chapters.

First a brief tour of the objects in the *Rum* world. A data domain and data operations are assumed given. Programs are expressions called *forms*. They are generated by constructions corresponding to the computation primitives in much the same way that well-formed formulas are generated by constructions corresponding to logical connectives. There are computation rules for each computation primitive (like deduction rules). To describe a particular computation, a form is closed in an *environment* that binds values to the free symbols. We call this closure a *dtree* or description tree to emphasize the relation between the tree structure of a form and that of the local structure of the computation described by the form. In addition to data, the computation domain contains *pfns* and *continuations.* Pfns are functional abstractions. They can be thought of as partial functions containing information describing how the value of the function is to be computed. This information consists of a symbol naming the argument, the pfn body – a form describing the computation, and the pfn environment – assigning values to the free symbols of the pfn body other than the argument symbol. Continuations are control abstractions. They represent computation contexts built up in the process of carrying out computations. A computation context contains the information describing how the computation is to continue when the current subcomputation returns a value. As objects of the computation domain they provide a means of remembering and switching contexts, and of suspending and resuming computations.

For this chapter we fix the data domain to be the S-expression domain built from atomic objects by pairing, where the set of atomic objects includes numbers. Number trees are those S-expressions built up from numbers. Thus a number tree e is either a number, or a pair of number trees $e_0 \bullet e_1$. For example, 2 and $(2 \bullet 3) \bullet 4$ are number trees. We will use n, n_0, ..., to stand for numbers and e, e_0, ..., to stand for number trees. The given operations on S-expressions include (using Lisp names) pairing and projection (Cons, Car, Cdr); tests for atoms and zero (Atom, ZeroP); and addition and multiplication (+, $*$).

II.1. Simple recursive computation of the tree product function

The tree product function $\pi(e)$ is defined by induction on number trees.

$$\pi(e) = \begin{cases} e & \text{if } e \text{ is an atom} \\ \pi(e_0) * \pi(e_1) & \text{if } e = e_0 \bullet e_1 \end{cases}$$

For example, $\pi((2 \bullet 3) \bullet 4) = 2 * 3 * 4$. A direct analog to this definition is the recursive definition of the pfn Tprod.

```
▷    Tprod(x) ← if(Atom(x), x, Tprod(Car(x)) * Tprod(Cdr(x)))
```

This definition can be interpreted as a recursive definition in the usual way. To compute Tprod′e, (read "Tprod applied to e"), first compute Atom′e. If true return e, otherwise compute Tprod′Car(e) and Tprod′Cdr(e) (using the definition of Tprod recursively) and then multiply the results.

The expression "Tprod′e" denotes a dtree – essentially the body of the pfn Tprod closed in an environment binding the pfn Tprod to the symbol Tprod and binding e to the argument symbol x. The sign ′ is used to distinguish between the dtree describing the computation of Tprod applied to a number tree and the resulting value, for which we use the usual notation Tprod(e). We use identifiers such as Car and Tprod (beginning with upper case letters) both to denote symbols and to denote the value we have assigned to that symbol. Values are assigned either as the name of a data operation as for Car or by a definition such as that given above for Tprod. Such symbols are said to be globally defined. Context will make it clear whether we are referring to the symbol or its value.

II.1.1. Generating a computation tree

The rules for computation are more than just rules for computing a value, they are also rules for generating computation trees. The value and much more information can be read off a completed computation tree. As an example we generate the computation tree for Tprod′(2 • 0). The initial stage, Figure 1 (Stage 0), is a tree with a single node labeled by the description of the computation to be carried out - Tprod′(2 • 0). This describes the computation of the pfn body in an environment binding 2 • 0 to x (we omit explicit mention of globally defined symbols such as Tprod).

$$\langle \mathsf{if(Atom(x), x, Tprod(Car(x)) * Tprod(Cdr(x)))} \mid \mathsf{x} \leftarrow 2 \bullet 0 \rangle$$

Below the initial node we add a node labeled by the description of test Atom′(2 • 0). This is a primitive computation that returns the empty sequence □ – our representation of false. This is indicated by adding $\hookrightarrow \Box$ to the label at the test node. Thus we have the situation shown in Figure 1 (Stage 1). Since the test returned false, the else branch is selected. A second node is added below the initial node and labeled $\eth_{\mathrm{app}}$.

$$\eth_{\mathrm{app}} = \langle \mathsf{Tprod(Car(x)) * Tprod(Cdr(x))} \mid \mathsf{x} \leftarrow 2 \bullet 0 \rangle$$

This is shown in Figure 1 (Stage 2). The sign $\twoheadrightarrow$ labeling the arc from the initial node to the else node is the *reduces-to* sign and signifies that the value of the parent node is the value of the else subcomputation.

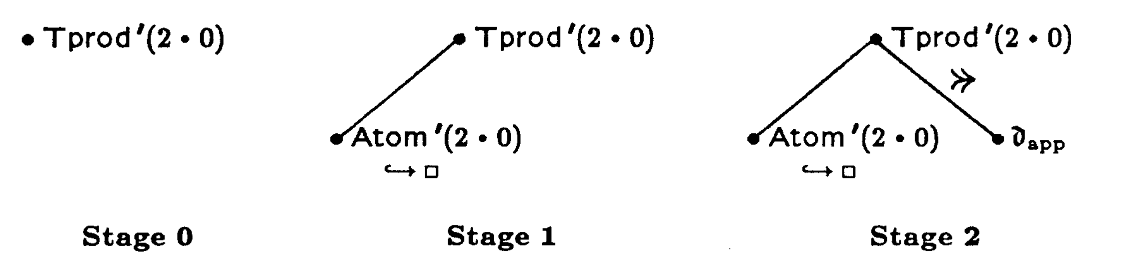
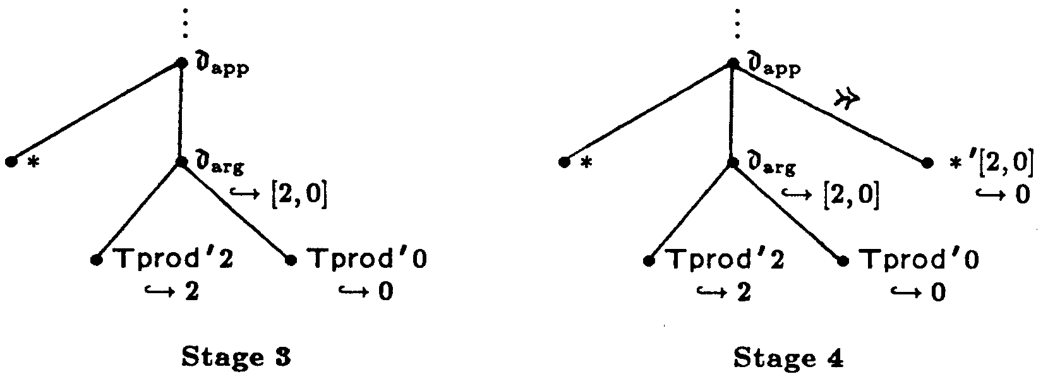


Figure 1. Initial stages in the computation of Tprod′(2 • 0)

Below the $\mathfrak{d}_{\text{app}}$ node we add a node labeled by $*$ – the function to be applied, and a node labeled by $\mathfrak{d}_{\text{arg}}$ which describes the computation of the argument sequence.

$$\mathfrak{d}_{\text{arg}} = \langle[\mathsf{Tprod}(\mathsf{Car}(x)), \mathsf{Tprod}(\mathsf{Cdr}(x))] \mid x \leftarrow 2 \cdot 0\rangle$$

Below the $\mathfrak{d}_{\text{arg}}$ node we generate the computation trees for $\mathsf{Tprod}'2$ and $\mathsf{Tprod}'0$. These trees may be generated in any order. That is, at each stage an extension can be made at any node at which a subcomputation can be started or a value returned. The value at the $\mathfrak{d}_{\text{arg}}$ node is the sequence of values returned by its two subcomputations. This is shown in Figure 2 (Stage 3). Now we add a third node below the $\mathfrak{d}_{\text{app}}$ node labeled by $*'[2,0]$ – the actual application of the function to the argument. This primitive computation returns the value 0. We have now reached the stage shown in Figure 2 (Stage 4).



Stage 3 **Stage 4**

Figure 2. Stages of the application subcomputation of $\mathsf{Tprod}'(2 \cdot 0)$

At this point, the computation is essentially complete, since there is a chain of $\succ\!\!\succ$ branches from the initial node to the node labeled $*'[2,0] \hookrightarrow 0$. What remains is to add value labels to the nodes along this chain. The complete tree is shown in Figure 3.

We can now elaborate the remark that Tprod corresponds directly to the inductive definition of π. For inductive definitions of the kind given for π there is, for each tuple in the defined relation, a unique derivation tree for that tuple obtained by using the clauses of the definition as derivation rules. The correspondence of Tprod and π is that the derivation tree for $\pi(e) = n$ has the same structure as the computation tree for $\mathsf{Tprod}'e \hookrightarrow n$.

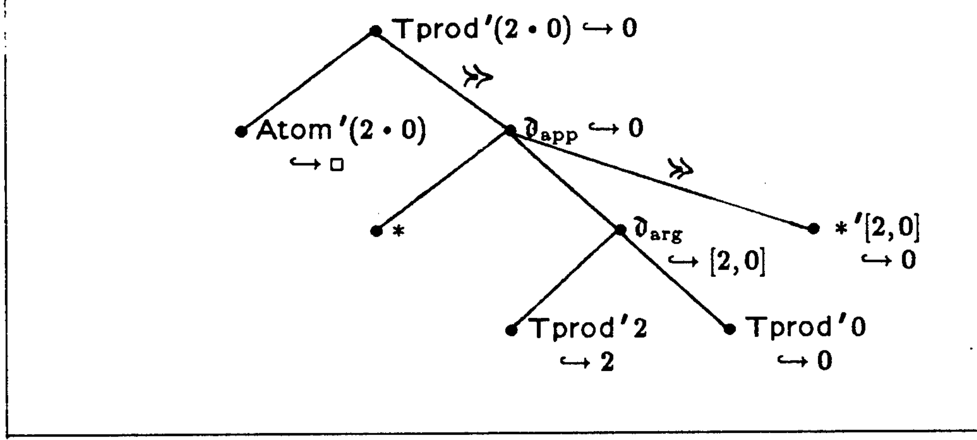


Figure 3. Computation tree for Tprod'(2 • 0)

II.1.2. Properties of Tprod

From the completed computation tree for Tprod'(2 • 0) we can read off the following information

$\mathsf{Tprod}'(2 \bullet 0) \hookrightarrow 0$

% Tprod applied to 2 • 0 returns value 0

$\mathsf{Tprod}'2 \prec \mathsf{Tprod}'(2 \bullet 0)$

% Tprod'2 is a subcomputation of Tprod'(2 • 0)

$\mathsf{Tprod}'(2 \bullet 0) \gg *'[2,0]$

% computation of Tprod'(2 • 0) reduces to computation of *'[2,0]

$count(\mathsf{Tprod}'(2 \bullet 0), \{\mathsf{Car}, \mathsf{Cdr}\}) = 2$

% Car and Cdr are applied two times in the computation of Tprod'(2 • 0)

$count(\mathsf{Tprod}'(2 \bullet 0), *) = 1$

% * is applied once in the computation of Tprod'(2 • 0)

The sign % flags remarks within formulas. $\hookrightarrow$ is the evaluation relation. In general $\mathfrak{d} \hookrightarrow v$ means that the computation described by $\mathfrak{d}$ returns the value v. The evaluation relation can be used to define the function computed by a pfn. $\prec$ is the subcomputation relation. $\mathfrak{d}_0 \prec \mathfrak{d}$ means that the computation described by $\mathfrak{d}_0$ is a subcomputation (a subtree) of the computation described by $\mathfrak{d}$. $\gg$ is the "reduces-to" relation which characterizes the main subcomputation in the case of conditional and application. The point is that to carry out the computation, one could simply replace the parent conditional or application node by its main subcomputation once this is determined.[1] $\ggcurly$ is analogous to "goto" while $\prec$ is analogous to "procedure call". For a finite set of data operations O, $count(\mathfrak{d}, O)$ is the number of applications of operations in O in the computation of $\mathfrak{d}$. For computation trees, $count(\mathfrak{d}, O)$ is the number of nodes in the computation tree for $\mathfrak{d}$ labeled by $o' d$ for some o in O and some argument sequence d.

Using the definition of Tprod, the rules for computation, and induction on number trees the following general facts can be derived.

$\mathsf{Tprod}' e \hookrightarrow \pi(e)$ % Tprod computes the tree product function

$e_0 <_{sexp} e \rightarrow \mathsf{Tprod}' e_0 \prec \mathsf{Tprod}' e$

$count(\mathsf{Tprod}' e, *) = cells(e)$

$count(\mathsf{Tprod}' e, \{\mathsf{Car}, \mathsf{Cdr}\}) = nodes(e) - 1$

where $<_{sexp}$ is the subtree relation on S-expressions, $cells(e)$ is the number of *cons*es used in the construction of e and $nodes(e)$ is the number of *cons*es plus the number of leaves in e.

We can also derive definitions of pfns computing properties of the computation tree for $\mathsf{Tprod}' e$ as a function of e. For example, MTprod computes the number of applications of $*$ in the computation of Tprod.

▷ $\mathsf{MTprod(x)} \leftarrow \mathsf{if(Atom(x), 0, 1 + MTprod(Car(x)) + MTprod(Cdr(x)))}$

and

$\mathsf{MTprod}' e \hookrightarrow count(\mathsf{Tprod}' e, *)$

II.2. Improvements of Tprod

Using properties of multiplication, we see that if one of the atoms of e is 0 then $\pi(e) = 0$. This suggests various possible modifications for improving the computation of $\pi(e)$. One such modification is

(†) *look for zeros first and apply* Tprod *only if the product is non-zero*

[1] We don't do this replacement because we wish the completed computation tree to be a complete record of the computation.

This computation is easy to describe. We define the pfn Inz to check for zeros in a number tree. Using Inz we define the pfn Tprod0 to describe the computation specified by (†).

```
▷        Inz(x) ← if(Atom(x), Zerop(x), or(Inz(Car(x)), Inz(Cdr(x))))
▷    Tprod0(x) ← if(Inz(x), 0, Tprod(x))
```

The form or(Inz(Car(x)), Inz(Cdr(x))) computes the disjunction of the two subexpressions. Inz(Car(x)) is computed first. If the result is true (a non-empty sequence) then true is returned without further computation. Otherwise the value of the disjunction is the value of Inz(Cdr(x)). From these definitions it is easy to see that

$$\mathsf{Tprod0}'e \hookrightarrow \pi(e)$$

$$\pi(e) = 0 \rightarrow count(\mathsf{Tprod0}'e, *) = 0$$

This modification is an improvement if multiplication is expensive and traversing the tree is cheap. If no zero is found, the number tree must be processed again. So, if the cost of traversing the tree is comparable to the cost of multiplication (for example the tree could be stored on disk, making execution of Car and Cdr expensive) then alternative improvements should be considered. For example

(‡) ***traverse the number tree, accumulating the product for completed subtrees, but terminate traversal and return zero if a zero is encountered.***

To make the requirement (‡) precise we define pfns Upto and Before. Upto(e) is the number of nodes visited before a zero is encountered in a left-first traversal of e. Before(e) is the number of non-atomic subtrees e_0 of e such that e_0 to the left of the left-most zero in e.

```
▷      Upto(x) ← if(Atom(x),
                    1,
                    if(Inz(Car(x)),
                       1 + Upto(Car(x)),
                       1 + Upto(Car(x)) + Upto(Cdr(x))))
▷    Before(x) ← if(Atom(x),
                    0,
                    if(Inz(Car(x)),
                       Before(Car(x)),
                       Before(Car(x)) + Before(Cdr(x)) + if(Inz(Cdr(x)), 0, 1)))
```

For example Upto(((2 • 3) • 0) • 4) = 6 and Before(((2 • 3) • 0) • 4) = 1.

A pfn p satisfying (‡) should only visit nodes until a zero is encountered and it should only do multiplications at nodes whose subtrees have been completely processed. In particular p should satisfy

(‡.*i*) $p' e \hookrightarrow \pi(e)$

(‡.*ii*) $count(p' e, \{\mathsf{Car}, \mathsf{Cdr}\}) = \mathsf{Upto}(e) - 1$

(‡.*iii*) $count(p' e, *) = \mathsf{Before}(e)$

We will give two solutions to this problem. For both solutions we define an auxiliary pfn *p.aux* which has an additional parameter used to store information about the state of the computation. The idea is to insure that as each node of the number tree is visited both the initial calling context and the context built up so far in traversing the number tree are available. Thus computation can continue normally or return zero immediately to the calling context. A pfn p computing π is defined by calling *p.aux* with a suitable initial value for the additional parameter.

II.2.1. Tprod1 – continuation-passing style computation

The first solution Tprod1 is based on continuation-passing style computation. The auxiliary for Tprod1 is Tprodc which has a continuation pfn as the additional argument and pfn formation is used to represent the computation context built up during traversal of a number tree. To continue the computation, the continuation pfn is applied to the value of the current subcomputation. The external context is always the calling context, thus a value is returned to the calling context by simply returning it. Tprod1 computes π by calling Tprodc with the identity pfn λ(z)z as the initial continuation pfn.

```
▷      Tprod1(x) ← Tprodc(x,λ(z)z)
▷    Tprodc(x,c) ← if(atom(x),
                        if(zerop(x),0,c(x)),
                        Tprodc(Car(x),λ(y)(Tprodc(Cdr(x),λ(z)c(y * z)))))
```

So far we have only discussed recursive definition as a means of generating pfns. The basic means of pfn formation is evaluation of a lambda-expression in an environment. For example the value of λ(z)c(y * z) is a pfn with argument symbol z, pfn body c(y * z) and pfn environment the evaluation environment. The application of a pfn to an argument is the closure of the pfn body in the pfn environment extended by binding the argument to the argument symbol.

One way to understand Tprodc is to note that it computes the same partial function as $\lambda(\mathsf{x},\mathsf{c})\mathsf{if}(\mathsf{Inz}(\mathsf{x}),0,\mathsf{c}(\mathsf{Tprod}(\mathsf{x})))$. We claim that the computation described by $\mathsf{Tprodc}'(e,\vartheta)$ returns zero immediately if a zero is encountered in the traversal of e and otherwise reduces to computation of $\vartheta'\pi(e)$. Thus Tprod1 satisfies ‡.

II.2.1.1. Analysis of the computation of Tprodc.

To verify the claim, we look in more detail at the computation described by $\mathsf{Tprodc}'(e,\vartheta)$. If $e = 0$ then 0 is returned as the value. If e is an atom and not zero, then computation reduces to application of ϑ to e – $\mathsf{Tprodc}'(e,\vartheta) \gg\!\!- \vartheta' e$. Otherwise computation reduces to application of Tprodc to $\mathsf{Car}(e)$ with continuation pfn given by $\lambda(\mathsf{y})(\mathsf{Tprodc}(\mathsf{Cdr}(\mathsf{x}),\lambda(\mathsf{z})\mathsf{c}(\mathsf{y} * \mathsf{z})))$ in an environment binding ϑ to c and e to x. In symbols

$$\mathsf{Tprodc}'(e,\vartheta) \gg\!\!- \mathsf{Tprodc}'(\mathsf{Car}(e),\vartheta_{\mathrm{a}}(e,\vartheta))$$

where

$$\vartheta_{\mathrm{a}} = \lambda(\mathsf{x},\mathsf{c})\lambda(\mathsf{y})(\mathsf{Tprodc}(\mathsf{Cdr}(\mathsf{x}),\lambda(\mathsf{z})\mathsf{c}(\mathsf{y} * \mathsf{z})))$$

If there are no zeros in $\mathsf{Car}(e)$, computation will eventually reduce to application of $\vartheta_{\mathrm{a}}(e,\vartheta)$ to $\pi(\mathsf{Car}(e))$, which further reduces to application of Tprodc to $\mathsf{Cdr}(e)$ with continuation pfn given by $\lambda(\mathsf{z})\mathsf{c}(\mathsf{y} * \mathsf{z})$ in an environment binding ϑ to c and $\pi(\mathsf{Car}(e))$ to y.

$$\mathsf{Tprodc}'(\mathsf{Car}(e),\vartheta_{\mathrm{a}}(e,\vartheta)) \gg\!\!- \vartheta_{\mathrm{a}}(e,\vartheta)'\pi(\mathsf{Car}(e)) \gg\!\!- \mathsf{Tprodc}'(\mathsf{Cdr}(e),\vartheta_{\mathrm{d}}(\pi(\mathsf{Car}(e)),\vartheta))$$

where

$$\vartheta_{\mathrm{d}} = \lambda(\mathsf{y},\mathsf{c})\lambda(\mathsf{z})\mathsf{c}(\mathsf{y} * \mathsf{z})$$

If there are no zeros in $\mathsf{Cdr}(e)$, computation will eventually reduce to the application of $\vartheta_{\mathrm{d}}(\pi(\mathsf{Car}(e)),\vartheta)$ to $\pi(\mathsf{Cdr}(e))$ which then reduces to application of ϑ to $\pi(e) = \pi(\mathsf{Car}(e)) * \pi(\mathsf{Cdr}(e))$.

$$\mathsf{Tprodc}'(\mathsf{Cdr}(e),\vartheta_{\mathrm{d}}(\pi(\mathsf{Car}(e)),\vartheta)) \gg\!\!- \vartheta_{\mathrm{d}}(\pi(\mathsf{Car}(e)),\vartheta)'\pi(\mathsf{Cdr}(e)) \gg\!\!- \vartheta'\pi(e)$$

Thus we see that if there is a zero in e then the reduction sequence terminates when the first one is encountered and zero is returned as the value, and otherwise $\mathsf{Tprodc}'(e,\vartheta) \gg\!\!- \vartheta'\pi(e)$. Hence $\mathsf{Tprod1}'e \hookrightarrow \pi(e)$. (Note that we have implicitly used induction on number trees in the above argument.)

II.2.1.2. Structure of computations in continuation-passing style.

The computation of $\mathsf{Tprodc}'(e,\vartheta)$ is a sequence of "reductions to" subcomputations determined by simple side computations. The result of the entire computation is the result of the final subcomputation. Thus the computation tree described by $\mathsf{Tprod1}'e$ is a sequential tree – a tree with a main (rightmost) branch where each subtree is connected by a "reduces-to" arc. For example the computation tree for $\mathsf{Tprod1}'(2 \bullet 0)$ has the shape (omitting side computations)

$$\begin{array}{l}
\bullet\mathsf{Tprod1}'(2 \bullet 0) \\
\quad \gg\!\!- \\
\quad \bullet\mathsf{Tprodc}'(2 \bullet 0, \lambda(z)z) \\
\qquad \gg\!\!- \\
\qquad \bullet\mathsf{Tprodc}'(2, \vartheta_{\mathrm{a}}(2 \bullet 0, \lambda(z)z)) \\
\qquad\quad \gg\!\!- \\
\qquad\quad \bullet\{\vartheta_{\mathrm{a}}(2 \bullet 0, \lambda(z)z))\}'2 \\
\qquad\qquad \gg\!\!- \\
\qquad\qquad \bullet\mathsf{Tprodc}'(0, \vartheta_{\mathrm{d}}(2, \lambda(z)z)) \\
\qquad\qquad\quad \gg\!\!- \\
\qquad\qquad\quad \bullet 0 \\
\qquad\qquad\quad\ \ \hookrightarrow 0
\end{array}$$

From the computation tree we see that

$$count(\mathsf{Tprod1}'2 \bullet 0, *) = 0 \quad \text{and} \quad count(\mathsf{Tprod1}'2 \bullet 0, \{\mathsf{Car}, \mathsf{Cdr}\}) = 2$$

More generally by the analysis of Tprodc given above, we can show that

$$count(\mathsf{Tprod1}'e, *) = \mathsf{Before}(e) \quad \text{and} \quad count(\mathsf{Tprod1}'e, \{\mathsf{Car}, \mathsf{Cdr}\}) = \mathsf{Upto}(e) - 1$$

Thus we have established that the pfn Tprod1 satisfies ‡.

II.2.2. Tprod2 – noting the calling context

The second solution Tprod2 uses continuation noting and resumption. The auxiliary pfn Tprodg has as additional argument a continuation representing the calling context. A value is returned to the calling context by applying the continuation parameter to that value. Otherwise the computation proceeds in the manner described by Tprod. Tprod2 *notes* its calling context and calls Tprodg with that continuation as the auxiliary parameter.[1]

```
▷        Tprod2(x) ← note(g)Tprodg(x,g)
▷     Tprodg(x,g) ← if(Atom(x),
                        if(Zerop(x),g(0),x),
                        Tprodg(Car(x),g) * Tprodg(Cdr(x),g))
```

In order to make sense of notions such as *noting* the current context and resuming a computation, we fix a definite order of evaluating subexpressions, namely left-most first. Thus a computation tree is generated by a uniquely determined sequence of stages. At each stage there is a current node and the context is the surrounding partial computation tree. The rules for noting and resumption are

(Note) To compute note(g)$\mathfrak{f}$ with continuation γ, compute $\mathfrak{f}$ with γ bound to g.

(Resume) If, in the process of carrying out a computation, a continuation γ is applied to a value v, the computation described by γ is resumed with v as the value returned by the subcomputation.

We say that a description returns a value normally if in any context the computation described returns a value to that context. Computations described by note-forms such as note(g)$\mathfrak{f}$ and by continuation application forms such as g($\mathfrak{f}_0$) when a continuation bound to g have the following properties.

(i) If $\mathfrak{f}$ returns a value normally then note(g)$\mathfrak{f}$ returns a value normally and the value returned by note(g)$\mathfrak{f}$ is the value of $\mathfrak{f}$.

(ii) If, in the process of evaluating $\mathfrak{f}$, a sub-expression g($\mathfrak{f}_0$) is evaluated and $\mathfrak{f}_0$ returns a value normally, then note(g)$\mathfrak{f}$ returns a value normally and the value of $\mathfrak{f}_0$ is the value of note(g)$\mathfrak{f}$.

(iii) More generally, if note(g)$\mathfrak{f}$ is evaluated with continuation γ, and either $\mathfrak{f}$ returns the value v normally or, during the computation of $\mathfrak{f}$, γ is applied to v then note(g)$\mathfrak{f}$ returns a value normally and v is the value returned by note(g)$\mathfrak{f}$.

(iv) If the value of g is a continuation and $\mathfrak{f}_0$, $\mathfrak{f}_1$ return values normally then the result of computing $\mathfrak{f}_1$(g($\mathfrak{f}_0$)) is the same as the result of computing g($\mathfrak{f}_0$).

In the computation described by Tprodg$'(e,\gamma)$, the continuation γ is carried as a parameter. If no zeros are encountered, the computation described by Tprodg$'(e,\gamma)$ is essentially that described by Tprod$'(e)$ and $\pi(e)$ is returned as a value. If a zero is encountered, zero is returned to γ, i.e. γ is applied to zero, which causes the

[1] note is similar to Reynold's **escape** construct and the Scheme **catch** construct.

computation to resume in the context represented by γ. Thus we see that $\mathsf{Tprod2}'e$ in any context returns value $\pi(e)$. This argument will be made more precise below as we explain the structure of computations such as those described by Tprodg.

II.2.2.1. Structure for computations noting continuations

Having fixed an order for evaluating subexpressions means that computation is a process of generating a sequence of computation stages. For sequential computation, we represent these stages by *states* which contain only the information from the corresponding stage needed to carry out the remainder of the computation. A state is a continuation γ, representing the computation context, together with a current subtask which is either to begin a subcomputation $\mathfrak{d}$ or to return a value v. In symbols a state has one of the forms $\gamma \triangledown \mathfrak{d}$ (begin computation of $\mathfrak{d}$ with continuation γ) or $\gamma \vartriangle v$ (return v to γ). The empty context is represented by the identity continuation Id. We say a computation sequence returns a value v if the last state in the sequence is $\mathsf{Id} \vartriangle v$. Continuations are built up from the identity continuation by composition of segments associated with nodes of the computation context along the path to the current node. In this sense continuations are similar to stack frames. Each segment describes the remainder of the computation at its associated node as a function of the value returned by the subcomputation in progress. For example $\mathsf{Appc}(\vartheta)$ is the continuation segment for an application node where the argument sequence is being computed. We write $\gamma \circ \mathsf{Appc}(\vartheta)$ for the composition of γ with $\mathsf{Appc}(\vartheta)$. If v is returned to $\gamma \circ \mathsf{Appc}(\vartheta)$, ϑ is applied to v with continuation γ. $\langle [\mathfrak{f}_0, \mathfrak{f}_1] \mid \xi \rangle$ describes an argument sequence. $\langle \mathsf{Carti}(\mathfrak{f}_1) \mid \xi \rangle$ is the continuation segment for an argument sequence node where the first argument $\langle \mathfrak{f}_0 \mid \xi \rangle$ is being computed. $\mathsf{Cartc}(v_0)$ is the continuation segment for an argument sequence node where the value of first argument is v_0 and the second argument is being computed. If v_0 is returned to $\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_1) \mid \xi \rangle$ then computation of $\langle \mathfrak{f}_1 \mid \xi \rangle$ is begun with continuation $\gamma \circ \mathsf{Cartc}(v_0)$. If v_1 is returned to $\gamma \circ \mathsf{Cartc}(v_0)$ then the sequence $[v_0, v_1]$ is returned to γ.

The computation described by $\mathsf{Tprod2}'e$ with continuation γ is a sequence of states beginning with $\gamma \triangledown \mathsf{Tprod2}'e$. $\mathsf{Tprod2}'e$ is $\mathsf{note(g)Tprodg(g,x)}$ in an environment binding e to x. According to the rule for note, the next step is to begin the computation described by $\mathsf{Tprodg}'(e, \gamma)$ with continuation γ. In symbols

$$\gamma \triangledown \mathsf{Tprod}'e = \gamma \triangledown \langle \mathsf{note(g)Tprodg(g,x)} \mid \mathsf{x} \leftarrowtail e \rangle \rightarrowtail \gamma \triangledown \mathsf{Tprodg}'(e, \gamma)$$

where $\rightarrowtail$ is the *step* relation on states. To compute $\mathsf{Tprodg}'(e, \gamma)$ with current continuation γ_{cur} there are three cases. If e is zero then $\mathsf{Atom}(e)$ and $\mathsf{Zerop}(e)$ are true and the conditional branch selected is $\mathsf{g}(0)$. Thus γ is applied to zero and computation resumes at the point represented by γ with zero as the value returned.

$$\gamma_{\mathrm{cur}} \triangledown \mathsf{Tprodg}'(0, \gamma) \rightarrowtail \gamma_{\mathrm{cur}} \triangledown \langle \mathsf{g}(0) \mid \mathsf{g} \leftarrowtail \gamma \rangle \rightarrowtail \gamma_{\mathrm{cur}} \triangledown \gamma' 0 \rightarrowtail \gamma \vartriangle 0$$

If e is atomic and non-zero the branch selected is x and e is returned to the current context.

$$\gamma_{\mathrm{cur}} \triangledown \mathsf{Tprodg}'(e,\gamma) \rightarrowtail \gamma_{\mathrm{cur}} \triangledown \langle \mathsf{x} \mid \mathsf{x} \leftarrow \mathsf{e}\rangle \rightarrowtail \gamma_{\mathrm{cur}} \vartriangle e$$

If e is non-atomic the branch selected is $*(\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g}),\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g}))$. Thus computation of the argument sequence

$$[\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g}),\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})]$$

is begun with the continuation extended by composition with $\mathsf{Appc}(*)$. In symbols, computation proceeds to the state

$$\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*) \triangledown \langle[\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g}),\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})] \mid \mathsf{x} \leftarrow e, \mathsf{g} \leftarrow \gamma\rangle$$

The next state begins computation of the first argument $\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g})$ with the continuation extended by composition $\langle\mathsf{Carti}(\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})) \mid \xi\rangle$. In symbols

$$\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*) \circ \langle\mathsf{Carti}(\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})) \mid \xi\rangle \triangledown \langle\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g}) \mid \xi\rangle)$$

If there are no zeros in $\mathsf{Car}(e)$, then eventually $\pi(\mathsf{Car}(e))$ will be returned and computation of $\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})$ will be begun. The continuation at this point is $\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*)$ composed with $\mathsf{Cartc}(\pi(\mathsf{Car}(e)))$

$$\ldots$$

$$\rightarrowtail$$

$$\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*) \circ \langle\mathsf{Carti}(\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})) \mid \xi\rangle \vartriangle \pi(\mathsf{Car}(e))$$

$$\rightarrowtail$$

$$\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*) \circ \mathsf{Cartc}(\pi(\mathsf{Car}(e))) \triangledown \langle\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g}) \mid \xi\rangle)$$

If there are no zeros in $\mathsf{Cdr}(e)$, then eventually $\pi(\mathsf{Cdr}(e))$ will be returned, $*$ will be applied to the pair of results, and $\pi(e)$ will be returned to γ_{cur}.

$$\ldots$$

$$\rightarrowtail$$

$$\gamma_{\mathrm{cur}} \circ \mathsf{Appc}(*) \circ \mathsf{Cartc}(\pi(\mathsf{Car}(e))) \vartriangle \pi(\mathsf{Cdr}(e))$$

$$\rightarrowtail$$

$$\gamma_{\mathrm{cur}} \triangledown *'[\pi(\mathsf{Car}(e)),\pi(\mathsf{Cdr}(e))]$$

$$\rightarrowtail$$

$$\gamma_{\mathrm{cur}} \vartriangle \pi(e)$$

As an example, the sequence of states for the computation of $\mathsf{Tprod2}'(2 \bullet 0)$ with continuation γ is (omitting some details)

$$
\begin{aligned}
&\gamma \triangledown \mathsf{Tprod2}'(2 \bullet 0) \\
&\quad \rightarrowtail \gamma \triangledown \mathsf{Tprodg}'(2 \bullet 0, \gamma) \\
&\quad \rightarrowtail \gamma_1 \triangledown \mathsf{Tprodg}'(2, \gamma) \\
&\quad \rightarrowtail \gamma_1 \vartriangle 2 \\
&\quad \rightarrowtail \gamma_2 \triangledown \mathsf{Tprodg}'(0, \gamma) \\
&\quad \rightarrowtail \gamma_2 \triangledown \gamma' 0 \\
&\quad \rightarrowtail \gamma \vartriangle 0
\end{aligned}
$$

where

$$
\begin{aligned}
\gamma_1 &= \gamma \circ \mathsf{Appc}(*) \circ \langle \mathsf{Carti}(\mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}), \mathsf{g})) \mid \mathsf{g} \leftarrow \gamma, \mathsf{x} \leftarrow 2 \bullet 0 \rangle \\
\gamma_2 &= \gamma \circ \mathsf{Appc}(*) \circ \mathsf{Cartc}(2)
\end{aligned}
$$

We read from the sequence of states

$$
count(\mathsf{Tprod2}' 2 \bullet 0, *) = 0 \quad \text{and} \quad count(\mathsf{Tprod2}' 2 \bullet 0, \{\mathsf{Car}, \mathsf{Cdr}\}) = 2
$$

Here $count(\mathsf{Tprod2}' e, O)$ is the number of states of the form $\gamma \triangledown o' d$ for some o in O and some data sequence d occurring in the computation sequence for $\mathsf{Tprod2}' e$.

The analysis above verifies our claim that Tprod2 computes the tree product function. Also we can show

$$
count(\mathsf{Tprod2}' e, *) = \mathsf{Before}(e) \quad \text{and} \quad count(\mathsf{Tprod2}' e, \{\mathsf{Car}, \mathsf{Cdr}\}) = \mathsf{Upto}(e) - 1
$$

Thus we have shown that Tprod2 satisfies ‡.

Remarks

- Both tree-structured computation and sequential computation are carried out as stepwise processes. The difference is that at each stage in the generation of a computation tree there is in general a frontier of active nodes where extensions can be made, while in the generation of a computation sequence there is a unique active node. A computation state can be thought of as the corresponding partial computation tree pruned and flattened in such a way that the active node is exposed.

- Tree-structured computations such as those described by Tprod, Inz, and Tprod1 can be carried out as sequential computations. The computation tree can be constructed from the completed computation sequence as continuations are essentially paths in the computation tree.

- In computations involving context switching, i.e. continuation application, the relations $\hookrightarrow$, $\prec$, $\gg\!\!-$ are not in general meaningful. For pfns such as Tprod2 that return a value normally, even though context switching may occur in the process of computation, the expressions $\mathsf{Tprod2}'e \hookrightarrow \pi(e)$ does make sense. Namely, it asserts that $\mathsf{Tprod2}'e$ returns $\pi(e)$ in any context.

II.2.3. Continuation-passing vs continuation-noting

By using continuation pfns, as in the Tprod1 example, we can remain within the world of tree-structured computations where pfns compute functions in the ordinary sense. On the other hand, using note, as in the Tprod2 example, the programs describing corresponding computations are much simpler, easier to understand (given a little practice) and easier to write. They are also more reliable, since the machine carrying out the computation constructs the continuation mechanically instead of the programmer constructing it by hand.

The analysis of the computations described by Tprodc and Tprodg involved essentially the same arguments applied to different representations of computation stages. In terms of abstract machines, Tprod1 and Tprod2 describe isomorphic computations, i.e. there is a correspondence between the computation steps. For example, the steps of $\mathsf{Tprodg}'2 \bullet 0$ correspond to those of $\mathsf{Tprodc}'2 \bullet 0$ as follows.

$$\begin{array}{ll}
\gamma \triangledown \mathsf{Tprod2}'(2 \bullet 0) & \mathsf{Tprod1}'(2 \bullet 0) \\
\rightarrowtail & \gg\!\!- \\
\gamma \triangledown \mathsf{Tprodg}'(2 \bullet 0, \gamma) & \mathsf{Tprodc}'(2 \bullet 0, \lambda(z)z) \\
\rightarrowtail & \gg\!\!- \\
\gamma_1 \triangledown \mathsf{Tprodg}'(2, \gamma) & \mathsf{Tprodc}'(2, \vartheta_a(2 \bullet 0, \lambda(z)z)) \\
\rightarrowtail & \gg\!\!- \\
\gamma_1 \vartriangle 2 & \vartheta_a(2 \bullet 0, \lambda(z)z)'2 \\
\rightarrowtail & \gg\!\!- \\
\gamma_2 \triangledown \mathsf{Tprodg}'(0, \gamma) & \mathsf{Tprodc}'(0, \vartheta_d(2, \lambda(z)z)) \\
\rightarrowtail & \gg\!\!- \\
\gamma \vartriangle 0 & 0
\end{array}$$

II.3. Equations and transformations

In this section we give some examples of equations and transformations. We first consider the case of tree-structured computation and begin with equations based on the value returned by a computation. We then look at some standard transformations on forms and at the corresponding transformations induced on the computations described. These tools are combined with program transformation methods to derive the recursion equation for Tprodc from an explicit definition given in terms of Tprod and Inz. Finally we show how many of the ideas for tree-structured computation can be extended to sequential computation. As an example of using program transformations involving continuations we also derive the recursion equation for Tprodg from an explicit definition given in terms of Tprod and Inz.

II.3.1. Equations for tree-structured computation

A simple characterization of the function computed by Tprodc is given by the equation

$$(\mathsf{Tprodc}\rightleftharpoons) \qquad \mathsf{Tprodc(x,c)} \rightleftharpoons \mathsf{if(Inz(x),0,c(Tprod(x)))}$$

where x ranges over number trees and c ranges over pfns. The meaning of this equation is that in any environment binding a number tree to x and a pfn to c, if the computation described by the form on either side of the $\rightleftharpoons$ sign returns a value, then both computations return the same value. This is another way of saying that Tprodc and $\lambda(\mathsf{x,c})\mathsf{if(Inz(x),0,c(Tprod(x)))}$ compute the same (partial) function on number trees and pfns. (Tprodc$\rightleftharpoons$) follows easily from the analysis of Tprodc given above.

One source of equations is recursive definition. For each recursive definition the equation obtained by replacing $\leftarrow$ by $\rightleftharpoons$ holds for all interpretations of the free symbols. For example

$$(\mathsf{Tprod}\triangleright) \qquad \mathsf{Tprod(x)} \rightleftharpoons \mathsf{if(Atom(x),x,Tprod(Car(x)) * Tprod(Cdr(x)))}$$

$$(\mathsf{Inz}\triangleright) \qquad \mathsf{Inz(x)} \rightleftharpoons \mathsf{if(Atom(x),Zerop(x),or(Inz(Car(x)),Inz(Cdr(x))))}$$

$$\begin{aligned}(\mathsf{Tprodc}\triangleright) \qquad \mathsf{Tprodc(x,c)} \rightleftharpoons \mathsf{if(}&\mathsf{atom(x),}\\ &\mathsf{if(zerop(x),0,c(x)),}\\ &\mathsf{Tprodc(Car(x),}\lambda\mathsf{(y)(Tprodc(Cdr(x),}\lambda\mathsf{(z)c(y * z)))))}\end{aligned}$$

$\rightleftharpoons$ is an equivalence relation, but not a true equality relation. Substitution of $\rightleftharpoons$-equivalent forms at a position in a given form yields $\rightleftharpoons$-equivalent forms iff that position is evaluated or discarded in the process of evaluating the entire form. We call such positions $\rightleftharpoons$-substitutable positions. For example we have

(i) $\mathsf{Tprod(x)} \rightleftharpoons \mathsf{if(lnz(x),0,Tprod(x))}$

(ii) $\mathsf{Tprod(x)} * \mathsf{y} \rightleftharpoons \mathsf{if(lnz(x),0,Tprod(x))} * \mathsf{y}$

(iii) $\neg(\lambda(\mathsf{y})(\mathsf{Tprod(x)} * \mathsf{y}) \rightleftharpoons \lambda(\mathsf{y})(\mathsf{if(lnz(x),0,Tprod(x))} * \mathsf{y}))$

(i) is just the assertion that Tprod0 computes the same function as Tprod. (ii) follows from (i) by substitution. The inequivalence (iii) is because the two form denote pfns which are the same as functions, but describe different computations and thus are different pfns.

II.3.2. Transformations for tree-structured computation

Now we look at some standard transformations on forms and the corresponding transformations induced on the computation trees described. $\mathfrak{f}$, $\mathfrak{f}_0$, ...stand for forms. $\mathfrak{f}_0 \mapsto \mathfrak{f}_1$ is read as "$\mathfrak{f}_0$ transforms to $\mathfrak{f}_1$".

If transformations

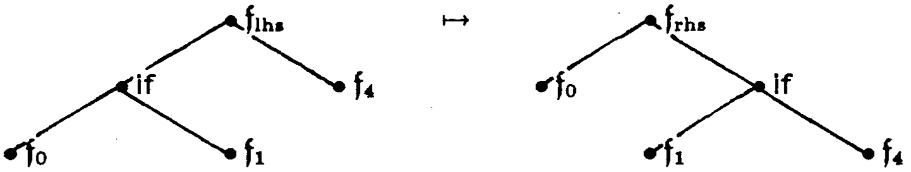
Two transformation rules for if-forms are if-distribution and if-simplification (introduced in McCarthy [1963b]). If-distribution has the form

(if.if) $\mathsf{if}(\mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2),\mathfrak{f}_3,\mathfrak{f}_4) \mapsto \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4))$

for distribution of if over if and

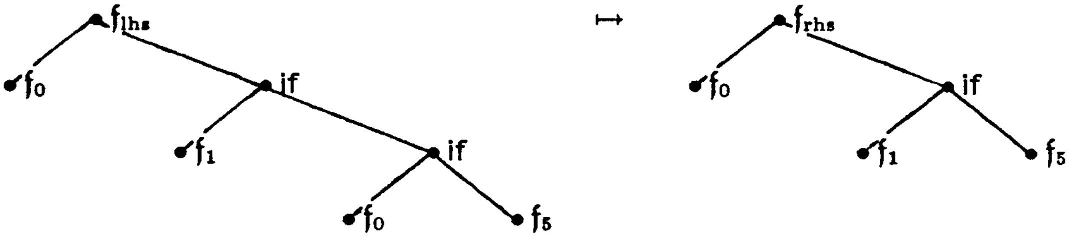
(if.ap) $\mathfrak{f}(\mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2)) \mapsto \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}(\mathfrak{f}_1),\mathfrak{f}(\mathfrak{f}_2))$

for distribution of application over if. We call the inverse of a distribution transformation a factoring. If $\mathfrak{f}_b$ is obtained from $\mathfrak{f}_a$ by application of if-distribution and factoring transformations at $\rightleftharpoons$-substitutable positions then $\mathfrak{f}_a \rightleftharpoons \mathfrak{f}_b$. The computation trees described by $\mathfrak{f}_a$ and $\mathfrak{f}_b$ are the same except for rearrangement of nodes where distribution or factoring has been applied. In particular, the amount of work in carrying out computations described by related forms is the same. For



$\mathfrak{f}_{\rm lhs} = \mathsf{if}(\mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2),\mathfrak{f}_3,\mathfrak{f}_4)$ $\mathfrak{f}_{\rm rhs} = \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4))$

Figure 4. The effect of if-distribution on computation structure.



$\mathfrak{f}_{\rm lhs} = \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4))$
where $\mathfrak{f}_4 = \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_5,\mathfrak{f}_6)$

$\mathfrak{f}_{\rm rhs} = \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_5),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_6))$

Figure 5. The effect of if-simplification on computation structure.

example, if $\mathfrak{f}_0$ is true and $\mathfrak{f}_1$ is false the computations of the left and right side of (if.if) have the shape shown in Figure 4.

A typical if-simplification is to omit redundant computations of a test. For example if $\mathfrak{f}_4 = \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_5,\mathfrak{f}_6)$ we have the simplification

(if.simp) $\mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4)) \mapsto \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_5),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_6))$

If $\mathfrak{f}_a$ transforms to $\mathfrak{f}_b$ by applying (if.simp) at $\rightleftharpoons$-substitutable positions then $\mathfrak{f}_a \rightleftharpoons \mathfrak{f}_b$ and the computations described by $\mathfrak{f}_b$ are related to those described by $\mathfrak{f}_a$ by pruning redundant subcomputations. Thus less work will generally be done computing $\mathfrak{f}_b$. For example, if $\mathfrak{f}_0$ is true and $\mathfrak{f}_1$ is false then the computations of the left and right side of (if.simp) have the shape shown in Figure 5.

A final if-simplification which we will need for our examples is (if.or) which eliminates disjunction in the test position.

$$\text{(if.or)} \qquad \mathsf{if}(\mathsf{or}(\mathfrak{f}_0,\mathfrak{f}_1),\mathfrak{f}_2,\mathfrak{f}_3) \mapsto \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_2,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_2,\mathfrak{f}_3))$$

(if.or) transformations applied at $\rightleftharpoons$-substitutable positions preserve $\rightleftharpoons$-equivalence.

Let conversions

Application expressions such as $\{\lambda(\mathsf{y})\mathfrak{f}_1\}(\mathfrak{f}_0)$ describe a special kind of pfn application which serves to make temporary bindings, i.e. to give a name to the value of an expression locally. To emphasize this fact, such expressions may be written $\mathsf{let}\{\mathsf{y} \leftarrow \mathfrak{f}_0\}\mathfrak{f}_1$ and read "let y be $\mathfrak{f}_0$ in $\mathfrak{f}_1$".

We call the substitution of $\mathfrak{f}_0$ for y in $\mathfrak{f}_1$ (with care taken not to trap free variables of $\mathfrak{f}_0$) let-elimination.

$$\text{(let.elim)} \qquad \mathsf{let}\{\mathsf{y} \leftarrow \mathfrak{f}_0\}\mathfrak{f}_1 \mapsto \mathfrak{f}_1|^{\mathsf{y}}_{\mathfrak{f}_0}$$

Let-elimination is analogous to β-reduction but must be restricted depending on the kind of relation we wish to preserve. For example to preserve definedness then either $\mathfrak{f}_0$ must be defined, or some occurrence of y in $\mathfrak{f}_1$ must be evaluated in the computation of $\mathfrak{f}_1$. Otherwise the transformed expression will be defined and the original will be undefined. To preserve $\rightleftharpoons$, all occurrences of y in $\mathfrak{f}_1$ must be evaluated or discarded in the course of evaluating $\mathfrak{f}_1$, i.e. y should not appear in an abstraction that is returned as a value since the substitution will change the value.

Forms related by let-eliminations describe computations which are related by replacing one evaluation of $\mathfrak{f}_0$, an application and some references to the symbol y by an evaluation of $\mathfrak{f}_0$ for each reference to the symbol y. If there is only one reference to y this reduces the work by eliminating the binding. If there are several references to y, and $\mathfrak{f}_0$ is a non-trivial expression, then the amount of work is increased by let-elimination. The inverse of let-elimination, let-introduction, is sometimes called common sub-expression elimination. We use let-conversion to refer to let-elimination and its inverse.

II.3.3. Derivation of the recursion equation for Tprodc

To illustrate the use of program transformations in *Rum* we outline the derivation of the recursion equation (Tprodc$\rhd$) for Tprodc from the equation (Tprodc$\rightleftharpoons$) defining the function computed by Tprodc. Such derivations can be used in two ways. If we assume Tprodc is defined by the (Tprodc$\rhd$) (i.e. by the corresponding recursive definition) the derivation of (Tprodc$\rhd$) from (Tprodc$\rightleftharpoons$) is a proof that Tprodc computes the function defined by (Tprodc$\rightleftharpoons$). Alternatively, if we assume that Tprodc is specified by (Tprodc$\rightleftharpoons$) then the derivation is an optimizing transformation by which the definition (Tprodc$\triangleright$) is obtained. Recall (§I.3) that unfolding means replacing an instance of lefthand side of an equation by the corresponding instance of the righthand side and folding means replacing an instance of righthand side of an equation by the corresponding instance of the lefthand side.

The main steps of the derivation follow. (Tprodc.a) is obtained from (Tprodc$\rightleftharpoons$) by unfolding using equations (Inz$\triangleright$) and (Tprod$\triangleright$) and then applying if- and or-simplifications.

$$\text{(Tprodc.a)}\qquad \begin{aligned}\mathsf{Tprodc}(c,x) \rightleftharpoons \mathsf{if}(&\mathsf{Atom}(x),\\ &\mathsf{if}(\mathsf{Zerop}(x),0,c(x)),\\ &\mathsf{if}(\mathsf{Inz}(\mathsf{Car}(x)),\\ &\quad 0,\\ &\quad \mathsf{if}(\mathsf{Inz}(\mathsf{Cdr}(x)),\\ &\qquad 0,\\ &\qquad c(\mathsf{Tprod}(\mathsf{Car}(x)) * \mathsf{Tprod}(\mathsf{Cdr}(x))))))\end{aligned}$$

Using let-conversion the final clause of (Tprodc.a) is expressed as the application of a pfn to Tprod(Cdr(x)).

$$\text{(Tprodc.b)}\qquad \begin{aligned}&c(\mathsf{Tprod}(\mathsf{Car}(x)) * \mathsf{Tprod}(\mathsf{Cdr}(x))) \rightleftharpoons\\ &\quad \{\lambda(z)c(\mathsf{Tprod}(\mathsf{Car}(x)) * z)\}(\mathsf{Tprod}(\mathsf{Cdr}(x)))\end{aligned}$$

Thus the final if-clause of (Tprodc.a) has the form of an instance of the right hand side of (Tprodc$\rightleftharpoons$) and by folding (Tprodc.c) is obtained.

$$\text{(Tprodc.c)}\qquad \begin{aligned}\mathsf{Tprodc}(c,x) \rightleftharpoons \mathsf{if}(&\mathsf{Atom}(x),\\ &\mathsf{if}(\mathsf{Zerop}(x),0,c(x)),\\ &\mathsf{if}(\mathsf{Inz}(\mathsf{Car}(x)),\\ &\quad 0,\\ &\quad \mathsf{Tprodc}(\mathsf{Cdr}(x),\lambda(z)c(\mathsf{Tprod}(\mathsf{Car}(x)) * z))\end{aligned}$$

Again using let-conversion the final clause of (Tprodc.c) is expressed as an instance of (Tprodc$\rightleftharpoons$)

$$\text{(Tprodc.d)}\qquad \begin{aligned}&\mathsf{Tprodc}(\mathsf{Cdr}(x),\lambda(z)c(\mathsf{Tprod}(\mathsf{Car}(x)) * z))\\ &\quad \{\lambda(y)\mathsf{Tprodc}(\mathsf{Cdr}(x),\lambda(z)c(y * z))\}(\mathsf{Tprod}(\mathsf{Car}(x)))\end{aligned}$$

and (Tprodc$\triangleright$) is obtained from (Tprodc.d) by folding.

II.3.4. Equations and transformations for sequential computation

Since $\hookrightarrow$ is not generally meaningful for sequential computation, neither is the equivalence $\rightleftharpoons$ as defined. However, there is a useful notion of equivalence obtained by replacing evaluation by "returns the same value in all contexts". This equivalence is denoted by $\approx$. The computation described by Tprodg is characterized explicitly in terms of lnz and Tprod by

$$(\mathsf{Tprodg}{\approx}) \qquad \mathsf{Tprodg}(\mathsf{x},\mathsf{g}) \approx \mathsf{if}(\mathsf{lnz}(\mathsf{x}),\mathsf{g}(0),\mathsf{Tprod}(\mathsf{x}))$$

where x ranges over number trees and g ranges over continuations. The meaning of the equation $(\mathsf{Tprodg}{\approx})$ is that for any environment ξ binding a number tree to x and a continuation to g, and for any continuation γ, if the computation sequence from the state beginning the computation described by the form on either side of the $\approx$ sign with continuation γ terminates with a value then both sequences terminate with the same value. In symbols

$$\gamma \triangledown \langle \mathsf{Tprodg}(\mathsf{x},\mathsf{g}) \mid \xi\rangle \rightarrowtail \mathsf{Id} \vartriangle v \leftrightarrow \gamma \triangledown \langle \mathsf{if}(\mathsf{lnz}(\mathsf{x}),\mathsf{g}(0),\mathsf{Tprod}(\mathsf{x})) \mid \xi\rangle \rightarrowtail \mathsf{Id} \vartriangle v$$

$(\mathsf{Tprodg}{\approx})$ follows easily from the analysis of the computation described by Tprodg given above. Furthermore, using the properties of note we have

$$\mathsf{Tprod}(\mathsf{x}) \approx \mathsf{note}(\mathsf{g})\mathsf{if}(\mathsf{lnz}(\mathsf{x}),\mathsf{g}(0),\mathsf{Tprod}(\mathsf{x}))$$

Two expressions that are $\rightleftharpoons$-equivalent as descriptions of computation trees are $\approx$-equivalent as descriptions of computation sequences. In particular, the equations for $\rightleftharpoons$ given above hold with $\rightleftharpoons$ replaced by $\approx$. Recursive definitions are also a source of $\approx$-equivalences. The equation obtained by replacing $\leftarrow$ in a recursive definition by $\approx$ holds for all interpretations of the free symbols. Thus

$$\begin{aligned}(\mathsf{Tprodg}{\rhd}) \quad \mathsf{Tprodg}(\mathsf{x},\mathsf{g}) \approx \mathsf{if}(&\mathsf{atom}(\mathsf{x}),\\ &\mathsf{if}(\mathsf{zerop}(\mathsf{x}),\mathsf{g}(0),\mathsf{x}),\\ &\mathsf{Tprodg}(\mathsf{Car}(\mathsf{x}),\mathsf{g}) * \mathsf{Tprodg}(\mathsf{Cdr}(\mathsf{x}),\mathsf{g})\end{aligned}$$

Substitution of $\approx$-equivalent expressions is limited to positions that are evaluated or discarded as for $\rightleftharpoons$.

Suitably restricted, the $\rightleftharpoons$-preserving transformations on forms such as if-distribution, if-simplification and let-elimination are also $\approx$ preserving. The restrictions are basically that subexpressions involved in a change in order of evaluation must return values normally. For (if.if) the expressions $\mathfrak{f}_0$,$\mathfrak{f}_1$,$\mathfrak{f}_2$ must return values normally; for (if.ap), the function expression $\mathfrak{f}$ and the test expression $\mathfrak{f}_0$

must return values normally; and for let-elimination (let.elim) the argument expression $\mathfrak{f}_0$ must return a value normally.

In addition there are transformations involving continuation application that preserve $\approx$. If g ranges over continuations and $\mathfrak{f},\mathfrak{f}_1$ return values normally then the context elimination transformation

$$\text{(ctxt.elim)} \qquad \mathfrak{f}(\mathsf{g}(\mathfrak{f}_1)) \mapsto \mathsf{g}(\mathfrak{f}_1)$$

is an equivalence preserving transformation. It also eliminates the work of computing the value of $\mathfrak{f}$. Combining context introduction with if-factoring we have

$$\mathsf{if}(\mathfrak{f}_0,\mathsf{g}(\mathfrak{f}_1),\mathfrak{f}(\mathfrak{f}_2)) \mapsto \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}(\mathsf{g}(\mathfrak{f}_1)),\mathfrak{f}(\mathfrak{f}_2)) \mapsto \mathfrak{f}(\mathsf{if}(\mathfrak{f}_0,\mathsf{g}(\mathfrak{f}_1),\mathfrak{f}_2))$$

is a $\approx$ preserving transformation when $\mathfrak{f}$, $\mathfrak{f}_0$, and $\mathfrak{f}_1$ return values normally.

II.3.5. Derivation of the recursion equation for Tprodg

Now we outline a derivation of the recursion equation (Tprodg$\triangleright$) from the explicit definition (Tprodg$\approx$). Since Inz and Tprod return values normally, the equations (Inz$\triangleright$) and (Tprod$\triangleright$) can be used for substitution and if- and let- transformations involving these pfns as tests and arguments are valid. Thus the basic ideas used in the derivation of the recursion equation for Tprodc apply to equations involving Tprodg. (Tprodg.a) is obtained from (Tprodg$\approx$) by unfolding using (Inz$\triangleright$) and (Tprod$\triangleright$) and then applying if-transformations.

$$\begin{array}{lll}
\text{(Tprodg.a)} \qquad & \mathsf{Tprodg}(\mathsf{x},\mathsf{g}) \approx & \mathsf{if}(\mathsf{Atom}(\mathsf{x}),\\
& & \quad \mathsf{if}(\mathsf{Zerop}(\mathsf{x}),\mathsf{g}(0),\mathsf{x}),\\
& & \quad \mathsf{if}(\mathsf{Inz}(\mathsf{Car}(\mathsf{x})),\\
& & \qquad \mathsf{g}(0),\\
& & \qquad \mathsf{if}(\mathsf{Inz}(\mathsf{Cdr}(\mathsf{x})),\\
& & \qquad\quad \mathsf{g}(0),\\
& & \qquad\quad \mathsf{Tprod}(\mathsf{Car}(\mathsf{x})) * \mathsf{Tprod}(\mathsf{Cdr}(\mathsf{x})))))
\end{array}$$

Using the fact that $\{\lambda(\mathsf{z})\mathfrak{f}\}\mathsf{g}(0) \approx \mathsf{g}(0)$ for any expression $\mathfrak{f}$ (context elimination) the context of Tprod(Cdr(x)) in the final if-clause of (Tprodg.a) can be factored out.

$$\begin{array}{ll}
\text{(Tprodg.b)} \quad & \mathsf{if}(\mathsf{Inz}(\mathsf{Cdr}(\mathsf{x})),\mathsf{g}(0),\mathsf{Tprod}(\mathsf{Car}(\mathsf{x})) * \mathsf{Tprod}(\mathsf{Cdr}(\mathsf{x})))\\
& \qquad \approx \{\lambda(\mathsf{z})(\mathsf{Tprod}(\mathsf{Car}(\mathsf{x})) * \mathsf{z})\}\mathsf{if}(\mathsf{Inz}(\mathsf{Cdr}(\mathsf{x})),\mathsf{g}(0),\mathsf{Tprod}(\mathsf{Cdr}(\mathsf{x})))
\end{array}$$

Matching the if-form on the righthand side to (Tprodg$\approx$) and folding (Tprodg.c) is obtained.

$$\text{(Tprodg.c)}\qquad \begin{array}{l} \mathsf{Tprodg(x,g)} \approx \mathsf{if(Atom(x),} \\ \qquad\qquad \mathsf{if(Zerop(x),g(0),x),} \\ \qquad\qquad \mathsf{if(Inz(Car(x)),} \\ \qquad\qquad\quad \mathsf{g(0),} \\ \qquad\qquad\quad \{\lambda\mathsf{(z)Tprod(Car(x))} * \mathsf{z}\}\mathsf{Tprodg(Cdr(x),g)))} \end{array}$$

Repeating these steps the context of Tprod(Car(x)) in the final if-clause of (Tprodg.c) is factored out and (Tprodg.d) is obtained by folding again with (Tprodg$\approx$).

$$\text{(Tprodg.d)}\qquad \begin{array}{l} \mathsf{Tprodg(x,g)} \approx \mathsf{if(Atom(x),} \\ \qquad\qquad \mathsf{if(Zerop(x),g(0),x),} \\ \qquad\qquad \{\lambda\mathsf{(y)}\{\lambda\mathsf{(z)y} * \mathsf{z}\}\mathsf{Tprodg(Cdr(x),g))}\} \\ \qquad\qquad\qquad \mathsf{Tprodg(Car(x)),g))} \end{array}$$

Applying let-conversions we obtain (Tprodg$\rhd$) as desired.

Chapter III. The meta-world

In this chapter we review the basic mathematical tools and structures used in constructing the *Rum* world and developing its theory. The main tools are constructions of sequences and finite maps, and inductive generation of domains, operations, and relations. Three kinds of structures play an important role in our work: *Rum* data structures, tree structures and binary relations over a given domain. We assume that the basic ideas are familiar ones. The following discussion is mainly to introduce our notation and conventions, and to point out the basic facts that we will use, generally implicitly. We have collected all the metamathematics into one chapter for convenience. It should be treated as reference material and need not be read in full before proceeding. In §1 the notion of function is discussed in order to clarify terminology. The material on finite sequences (§2), finite maps (§3), and finite forms of inductive generation (§4) is used from the beginning and the reader should be familiar with the notions presented in these sections. The class of *Rum* data structures is described in §5 and two examples are given: the trivial structure and the S-expression structure. Only the basic features of *Rum* data structures are used for most of the work in *Rum*. The S-expression structure serves as a concrete and typical example. It is the structure used informally in Chapter II and will be used in examples given in Chapter IV and Appendix B. Tree structures occur throughout the work in *Rum*. For the most part we think of tree structures pictorially as tree-shaped graphs with additional labeling information associated with some nodes. The point is to provide geometric intuitions about these objects and certain kinds of operations on them. The material presented in the section on tree structures as mathematical structures (§6) is used only in Appendix A and Appendix B. The material presented in the section on binary relations (§7) is used mainly in Chapter VI.

As to general format, references to chapter will be by number (upper case roman numerals). References to sections within the same chapter are by section number while references to sections in other chapters contain the chapter number and the section number. Thus §3 refers to section 3 of the current chapter and §IV.2 refers to section 2 of chapter IV. Definitions of constants, functions, relations, etc. are marked by the sign ▷. For example this is the format of the pfn definitions given in Chapter II. Facts, lemmas, theorems, etc. are marked by the sign ■. Equations, theorems, and other such items to which we may wish to refer are given names. For displayed items, the name appears at the side of the item. This is the format used for naming transformations in §II.3.

■ **A fact** ...

Gives the name "a fact" to the fact described by ....

We will mostly work within an informal set-theoretic framework. $\mathbb{N}$ denotes the natural numbers and we will use i, j, m, n to range over natural numbers. We use braces {, } in several different kinds of expressions – in application expressions for readability, for set formation, substitution into a context, Kleene braces, etc. Many of these uses have already appeared in Chapter I. They will be pointed out again when they first arise officially. There should be no confusion, as context will always determine what use is intended.

Formal theories in which various fragments of our work can be represented quite naturally are given in Feferman [1979, 1981, 1982]. In particular, the definitions of the domains and basic computation relations of $\mathcal{R}um$ can be carried out in FM0 [Feferman 1982] (a theory of inductive definitions with the proof-theoretic strength of primitive recursive arithmetic). The entire work of chapters IV, V, and VII and the examples of Appendix B can be carried out in FM1 [Feferman 1982] (an extension of FM0 with proof-theoretic strength of Peano arithmetic). The work on comparison relations (Chapter VI) requires stronger theories such as Feferman [1979] or Feferman [1981]. These latter theories treat functions and classes as intensional objects. It is unclear how far one can go in such theories in formalizing some aspects of extensionality used in discussing comparisons, however this is a very small part of the work on comparisons.

We will sometimes use the terms *informal* and *formal* loosely to distinguish between ideas presented by picture or example and ideas presented using precisely defined structures and relations. It will generally be the case that formal in the sense just explained will be a large step towards formal in the logical sense of representation within a formal system.

III.1. What is a function?

As pointed out in Chapter I, there are two interpretations of "function". Both involve the idea that a function prescribes a map from its domain A to its range B. One interpretation (the intensional view) is that a function is a rule for obtaining a value in B given an argument in A. The other interpretation (the extensional view) is that a function is a graph, i.e. the set of pairs (a, b) in $A \times B$ such that a is mapped to b by the function. Functions were originally thought of as rules and this view is maintained in constructive mathematics. In recursion theory and model theory the extensional view is generally taken. In computer science literature both interpretations are found and the term is often used quite ambiguously.

For many purposes, if one maintains a consistent view, it does not matter which view is taken. The differences become important when one considers such matters as

(i) when are two functions to be considered equal

(ii) what is the space of functions from A to B for given domains A and B.

While ambiguity is not always a defect, in *Rum* we must say what we mean, as we are interested in questions such as (i) and (ii). In order to conform with modern usage in logic and mathematics we will use "function" in the extensional sense. We use $[A \to B]$ to denote the space of total functions from A to B and $[A \rightsquigarrow B]$ to denote the space of partial functions from A to B. For $f_0, f_1 \in [A \rightsquigarrow B]$, $f_0 \sqsubseteq f_1$ means that f_1 is an extension of f_0, i.e. the domain of f_0 is contained in the domain of f_1 and f_0 and f_1 agree on the domain of f_0. In other words, $\sqsubseteq$ is the subset relation graphs of partial functions.

III.2. Finite sequences

Finite sequences are a means of packaging a finite collection of things in a given order. For a given domain A, A^* is the domain of sequences from A. $\Box$ is the empty sequence, it 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 from singleton sequences. If a is an element of A and v is a sequence then $[a,v]$ is a non-empty sequence; $1^{\mathrm{st}}[a, v]$ is a, the first element; and $\mathrm{r}^{\mathrm{st}}[a, v]$ is v, the remainder. $1^{\mathrm{st}}\Box$ is $\Box$ and $\mathrm{r}^{\mathrm{st}}\Box$ is $\Box$. For $i < |v|$, $v{\downarrow}_i$ is the i^{th} element of v. In particular, $v{\downarrow}_0 = 1^{\mathrm{st}}\, v$ and $v{\downarrow}_{i+1} = (\mathrm{r}^{\mathrm{st}}\, v){\downarrow}_i$. Concatenation is associative with the empty-sequence as right and left identity. We write $[v_1, \ldots, v_n]$ for the concatenation of the sequences $v_1, \ldots, v_n$. 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. Since A^* is generated from the empty sequence and elements of A by concatenation we may make definitions and proofs by sequence induction. Figure6 summarizes the facts about sequences.

Generation of A^* from A

$$\square \in \mathsf{A}^*, \qquad a \in \mathsf{A}^*, \qquad [v_0, v_1] \in \mathsf{A}^*$$

Laws for sequences

$$[[v_0, v_1], v_2] = [v_0, [v_1, v_2]] \qquad [v_0, \square] = [\square, v_0] = \square$$

$$|\square| = 0 \qquad |a| = 1 \qquad |[v_0, v_1]| = |v_0| + |v_1|$$

$$1^{st}\,[a, v] = a \qquad 1^{st}\,\square = \square \qquad r^{st}\,[a, v] = v \qquad r^{st}\,\square = \square$$

$$v{\downarrow}_0 = 1^{st}\,v \qquad v{\downarrow}_{i+1} = (r^{st}\,v){\downarrow}_i \qquad a \in v \leftrightarrow (\exists v_0, v_1) v = [v_0, a, v_1]$$

where a is an element of A and $v\,,v_0$ ... are elements of A^*

Figure 6. About finite sequences

III.3. Finite maps

Finite maps are functions with finite domains that are presented as finite sets of argument value pairs called bindings. For domains A_0 and A_1, $[\mathsf{A}_0 \ast\!\!\succ \mathsf{A}_1]$ is the set of finite maps from A_0 to A_1, generated from the empty map by the binding operation. The empty map $\{\}$ is given by the empty set and has an empty domain. For $a_0 \in \mathsf{A}_0$, $a_1 \in \mathsf{A}_1$, and ξ a finite map from A_0 to A_1, $\xi\{a_0 \leftarrow a_1\}$ is the map obtained from ξ by binding a_1 to a_0. The domain of $\xi\{a_0 \leftarrow a_1\}$ is obtained from the domain of ξ by adding a_0. For a in the domain of $\xi\{a_0 \leftarrow a_1\}$ we have

$$\xi\{a_0 \leftarrow a_1\}(a) = \begin{cases} a_1 & \text{if } a = a_0 \\ \xi(a) & \text{if } a \neq a_0. \end{cases}$$

Two finite maps are equal just when they are equal as functions. Different constructions may give rise to the same functions since old bindings are forgotten.

Given a distinguished (default) element $a_\star$ of A_1 each finite map determines a total function from A_0 to A_1 which maps elements not in the domain of the finite map to $a_\star$. We use $[\mathsf{A}_0 \ast\!\!\succ \mathsf{A}_1]_{a\star}$ to indicate that a distinguished element $a_\star$ has been fixed. When A_1 is a sequence domain we often take the empty sequence as the distinguished element.

The rules for generation of finite maps from A_0 to A_1 and for application of finite maps are summarized in Figure 7.

Generation of $[\mathbb{A}_0 \ast\!\!\succ \mathbb{A}_1]$

$$\{\} \in [\mathbb{A}_0 \ast\!\!\succ \mathbb{A}_1], \qquad \xi\{a_0 \leftarrow\!\!\!\!+ a_1\} \in [\mathbb{A}_0 \ast\!\!\succ \mathbb{A}_1]$$

The domain of a finite map

$$dom(\{\}) = \emptyset \qquad dom(\xi\{a_0 \leftarrow\!\!\!\!+ a_1\}) = dom(\xi) \cup \{a_0\}$$

The function associated with a finite map with default $a_\star$

$$\{\}(a_0) = a_\star \qquad \xi\{a_0 \leftarrow\!\!\!\!+ a_1\}(a) = \begin{cases} a_1 & \text{if } a = a_0 \\ \xi(a) & \text{if } a \neq a_0 \end{cases}$$

Equality of finite maps with default

$$\xi_0 = \xi_1 \;\leftrightarrow\; (\forall a \in dom(\xi_0) \cup dom(\xi_1))(\xi_0(a) = \xi_1(a))$$

where $\xi, \xi_0, \xi_1 \in [\mathbb{A}_0 \ast\!\!\succ \mathbb{A}_1]$ and $a, a_0 \in \mathbb{A}_0$, $a_1 \in \mathbb{A}_1$

Figure 7. About finite maps

III.4. Finite inductive generation of objects and relations

Finite inductive generation is our main tool for defining domains, operations and relations. A finitely generated domain is given by a set of rules for constructing objects in the domain. The interpretation is that the only objects in the domain are those obtained by a finite sequence of constructions. The generation of the finite map domain over domains $\mathbb{A}_0$, $\mathbb{A}_1$ is an example of our presentation of finitely generated domains. The rules for constructions are expressions which implicitly contain the information as to the construction operation and its type. In addition, rules for equality may given. We assume that objects generated by different constructions (named by different terms) are different unless they can be proved equal using the rules for equality. Abstractly the domains we consider are freely generated algebras modulo an equivalence relation, i.e. they are initial algebras. The signature of the algebra can be read from the construction rules. For inductively generated domains we have principles for definition and proof by induction on the generation of objects in the domain. For example the application laws for finite maps and the definition of the domain of a finite map are defined by finite map induction.

Many of the relations defined in our world are given by inductive definition as well. In the finitary case we present such definitions by giving a set of rules (formulas) 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. 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.

III.5. *Rum* data structures

We are concerned with computation over data structures belonging to a class called generalized algebraic data structures. Such structures consist of a data domain and a set of operations that act on sequences from the domain. It is the latter that distinguishes these structures from ordinary algebraic structures.

We use $\mathfrak{D}$ (plain or subscripted) to denote a data structure. We assume that $\mathfrak{D}$ is given as a triple $\langle \mathbb{D}, \mathbb{O}, ap \rangle$ where $\mathbb{D}$ is the data domain; $\mathbb{O}$ is a set of (codes for) data operations; and ap is the application operation which applies data operations to sequences of data. If o is a data operation and $[a_1, \ldots, a_{\rm n}]$ is a sequence from the data domain, then $ap(o, [a_1, \ldots, a_{\rm n}])$ is the result of applying the operation coded by o to $[a_1, \ldots, a_{\rm n}]$. We will generally omit explicit mention of the application operation and use ordinary applicative notation, writing $o[a_1, \ldots, a_{\rm n}]$ for $ap(o, [a_1, \ldots, a_{\rm n}])$. We can think of operations having a variable number of arguments, or as having a single argument which is a sequence of arbitrary finite length.

III.5.1. The trivial data structure

One data structure of interest is the structure in which the data domain is empty and the only operation is Triv, which maps the empty sequence to the empty sequence. We will denote this structure by $\mathfrak{D}_\emptyset$. Thus

$$\mathfrak{D}_\emptyset = \langle \emptyset, \{\mathsf{Triv}\}, \{(\mathsf{Triv}, \Box, \Box)\} \rangle.$$

Sort	Constructor	Constructor Domain	Recognizer	Unconstructor	Notation
$\mathbb{D}_{zero}$	ZeroMk	$\{\}^*$	ZeroP		$\mathsf{ZeroMk}(\Box) = 0$
$\mathbb{D}_{neg}$	Sub1	$\mathbb{D}_{neg} \oplus \mathbb{D}_{zero}$	NegP	Add1	$\mathsf{Sub1}(z) = z - 1$
$\mathbb{D}_{pos}$	Add1	$\mathbb{D}_{pos} \oplus \mathbb{D}_{zero}$	PosP	Sub1	$\mathsf{Add1}(z) = z + 1$
$\mathbb{D}_{str}$	StrMk	$\mathbb{D}_{int}^*$	StrP	StrUn	$\mathsf{StrMk}[z_1, \ldots, z_n]$ $= "z_1, \ldots, z_n"$
$\mathbb{D}_{mtl}$	MtlMk	$\{\}^*$	MtlP		$\mathsf{MtlMk}(\Box) = \mathsf{Mtl}$
$\mathbb{D}_{pair}$	PairMk	$\mathbb{D}_{sexp} \otimes \mathbb{D}_{sexp}$	PairP	PairUn	$\mathsf{PairMk}[a_1, a_2]$ $= a_1 \bullet a_2$

where

$z, z_1, \ldots z_n$ **ranges over** $\mathbb{D}_{int} = \mathbb{D}_{neq} \oplus \mathbb{D}_{zero} \oplus \mathbb{D}_{pos}$

$a, a_1, \ldots a_n$ **ranges over** $\mathbb{D}_{sexp} = \mathbb{D}_{int} \oplus \mathbb{D}_{str} \oplus \mathbb{D}_{mtl} \oplus \mathbb{D}_{pair}$

Lists: $<>$; $a \bullet <a_1, \ldots, a_n> = <a, a_1, \ldots, a_n>$

Figure 8. The S-expression data structure

III.5.2. The S-expression data structure

The S-expression data structure $\mathfrak{D}_{sexp} = \langle \mathbb{D}_{sexp}, \mathbb{O}_{sexp}, ap_{sexp} \rangle$ 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. S-expression operations and notation are summarized in Figure 8.

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

The S-expression operations $\mathbb{O}_{sexp}$ are constructors, unconstructors, and recognizers for each of the sorts. $\mathbb{D}_{sexp}$ is freely generated by the construction operations applied to suitable sequences of S-expressions. $\mathbb{D}_{zero}$ is generated by ZeroMk applied to the empty sequence; $\mathbb{D}_{neg}$ is generated by Sub1 applied to non-positive integers; and $\mathbb{D}_{\mathrm{pos}}$ is generated by Add1 applied to non-negative integers. $\mathbb{D}_{\mathrm{mtl}}$ is generated by MtlMk applied to the empty sequence; $\mathbb{D}_{\mathrm{str}}$ is generated by StrMk applied to sequences of integers; and $\mathbb{D}_{\mathrm{pair}}$ 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 $\mathbb{D}_{\mathrm{neg}}$ is NegP; for $\mathbb{D}_{\mathrm{zero}}$ is ZeroP; for $\mathbb{D}_{\mathrm{pos}}$ is PosP; for $\mathbb{D}_{\mathrm{str}}$ is StrP; for $\mathbb{D}_{\mathrm{mtl}}$ is MtlP; and for for $\mathbb{D}_{\mathrm{pair}}$ 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.

We use the usual notation for integers, $\ldots\ -2,-1,0,1,2,\ \ldots$. We write $"z_0,\ldots,z_m"$ for $\mathsf{StrMk}[z_0,\ldots,z_m]$ and $a \bullet b$ for $\mathsf{PairMk}[a,b]$. The following equations illustrate the laws for data operations:

$$\begin{array}{ll}
\mathsf{StrMk}[z_1,\ldots,z_n] = "z_1,\ldots,z_n" & \mathsf{PairP}("z_1,\ldots,z_n") = \square \\
\mathsf{StrP}("z_1,\ldots,z_n") = "z_1,\ldots,z_n" & \mathsf{PairUn}("z_1,\ldots,z_n") = \square \\
\mathsf{StrUn}("z_1,\ldots,z_n") = [z_1,\ldots,z_n] & \mathsf{Add1}("z_1,\ldots,z_n") = \square
\end{array}$$

Lists are the subset of S-expressions generated from the empty list by pairing arbitrary S-expressions with lists. <> is the empty list and we write $<a_1\ \ldots\ a_n>$ for $a_1 \bullet (\ldots\ (a_n \bullet <>)\ldots)$. Lists are an important data structure for symbolic computation. They provide a natural encoding for both sequences and tree structures.

Remarks

- Our version of the S-expressions has usual data domain; what is different is our presentation and choice of primitive operations.
- $\mathfrak{D}_{\mathrm{sexp}}$ is the "standard" encoding domain for the metamathematics of $\mathcal{R}um$.

III.6. Tree structures

Although intuitively we think of tree structures in terms of pictures, it is useful to provide mathematical structures having the essential properties of these intuitive structures. We formalize tree structures using *tree domains* and *labeling functions.* A tree domain Δ is a set of sequences of numbers (called nodes) such that if $[\nu, j]$ is in Δ then ν is in Δ. $\mathbb{T}$ is the set of tree domains. A tree domain may be finite or infinite. $|\Delta|$ is the cardinality of Δ. A tree structure of type $<\mathsf{A}_1, \ldots, \mathsf{A}_n>$ is a tuple $<\Delta, f_1, \ldots, f_n>$ where Δ is a tree domain and, for $1 \leq i \leq n$, f_i is a function with domain contained in Δ and range A_i. $\mathbb{T} : <\mathsf{A}_1, \ldots, \mathsf{A}_n>$ is the set of tree structures of type $<\mathsf{A}_1, \ldots, \mathsf{A}_n>$. Thus we have

$$\mathbb{T} \underset{\text{df}}{=} \{\Delta \subset \mathbb{N}^* \mid [\nu, j] \in \Delta \rightarrow \nu \in \Delta\}$$

$$\mathbb{T} : <\mathsf{A}_1, \ldots, \mathsf{A}_n> \underset{\text{df}}{=}$$

$$\{<\Delta, f_1, \ldots f_n> \mid \Delta \in \mathbb{T} \wedge f_1 \in [\Delta \rightsquigarrow \mathsf{A}_1] \wedge \ldots \wedge f_n \in [\Delta \rightsquigarrow \mathsf{A}_n]\}$$

For $\tau = <\Delta, f_1, \ldots, f_n>$, Δ is called the domain of τ and f_i the i-th labeling function.

There is a simple correspondence between tree structures and pictures of trees such as the computation stages shown in §II.1 (Figures 1, 2, 3). The elements of the tree domain code paths to nodes in a tree shaped graph, and each labeling function associates a label to the nodes of its domain.

The main operations and relations on tree structures which are of interest for our work are the subtree operation, replacement, union of a set of compatible trees, and the containment relation. In the following, let $\tau = <\Delta, \ldots, f_i, \ldots>$, $\tau_0 = <\Delta_0, \ldots, f_{0,i}, \ldots>, \ldots$ be tree structures of type $<\mathsf{A}_1, \ldots, \mathsf{A}_n>$. It is easy to see from the definitions that the operations give tree structures of type $<\mathsf{A}_1, \ldots, \mathsf{A}_n>$.

▷ **Subtree.** For ν in Δ, the subtree at ν in τ (written $\tau{\downarrow}_\nu$) is defined by

$$\tau{\downarrow}_\nu = <\Delta{\downarrow}_\nu \ldots f_i{\downarrow}_\nu \ldots>$$

where

$$\Delta{\downarrow}_\nu = \{\nu_0 \mid [\nu, \nu_0] \in \Delta\} \quad \text{and} \quad f_i{\downarrow}_\nu(\nu_0) = \begin{cases} f_i([\nu, \nu_0]) & \text{if } [\nu, \nu_0] \in \Delta \\ \text{undefined} & \text{otherwise} \end{cases}$$

▹ **Replacement.** For ν in Δ, the result of replacing the subtree at ν by τ_0 (written $\tau\{\nu \nleftarrow \tau_0\}$) is defined by

$$\tau\{\nu \nleftarrow \tau_0\} = <\Delta\{\nu \nleftarrow \Delta_0\}, \ldots, f_i\{\nu \nleftarrow f_{0,i}\}, \ldots>$$

where

$$\Delta\{\nu \nleftarrow \Delta_0\} = \{[\nu, \nu_0] \mid \nu_0 \in \Delta_0\} \cup (\Delta - \Delta{\downarrow}_\nu)$$

and

$$f_i\{\nu \nleftarrow f_{0,i}\}(\nu_1) = \begin{cases} f_{0,i}(\nu_0) & \text{if } \nu_1 = [\nu, \nu_0] \wedge \nu_0 \in \Delta_0 \\ f_i(\nu_1) & \text{if } \nu_1 \in (\Delta - \Delta{\downarrow}_\nu) \\ \text{undefined} & \text{otherwise} \end{cases}$$

▹ **Containment.** τ_0 is contained in τ (written $\tau_0 \sqsubseteq \tau$) if Δ_0 is a subset of Δ, and $f_{0,i} \sqsubseteq f_i$.

▹ **Union.** Let T be a set of tree structures of type $<\mathbb{A}_1, \ldots, \mathbb{A}_n>$. T is compatible if for each pair of trees in T, corresponding pairs of labeling functions agree on their common domain. For such T we define the union ($\cup T$) to be the minimal tree structure containing every element of T.

$$\cup T = <\Delta_{\cup T}, \ldots, f_{\cup T,i}, \ldots>$$

where

$$\Delta_{\cup T} = \{\nu \mid (\exists <\Delta \ldots, f_i, \ldots> \in T)(\nu \in \Delta)\}$$

$$f_{\cup T,i}(\nu) = \begin{cases} f_i(\nu) & \text{if } <\Delta \ldots, f_i, \ldots> \in T \quad \text{and} \quad \nu \in dom(f_i) \\ \text{undefined} & \text{if } (\forall <\Delta \ldots, f_i, \ldots> \in T)(\nu \notin dom(f_i)) \end{cases}$$

Note that compatibility assures that this definition determines a well defined tree structure, and $\tau \in T$ implies $\tau \sqsubseteq \cup T$.

III.7. About binary relations

This section contains a summary of basic notions about binary relations. Let $\mathbb{A}$ be a domain, $\mathbb{A}_0$ a subdomain of $\mathbb{A}$. Binary relations on $\mathbb{A}$ are subsets of $\mathbb{A} \times \mathbb{A}$, and as such, inherit the operations and properties of sets of pairs. We will generally write $a_0 \; \rho \; a_1$ for $(a_0, a_1) \in \rho$. The following is a summary of definitions of standard operations and relations on binary relations that we will need for our work. a, a_0, a_1 range over elements of $\mathbb{A}$; ρ, ρ_0, ρ_1 range over binary relations on $\mathbb{A}$. Γ denotes a set of binary relations on $\mathbb{A}$.

▹ **Subrelation:** $\rho_0 \subset \rho_1 \leftrightarrow a_0 \; \rho_0 \; a_1 \rightarrow a_0 \; \rho_1 \; a_1$

▹ **Suprelation:** $\rho_0 \supset \rho_1 \leftrightarrow \rho_1 \subset \rho_0$

▹ **Equality:** $\rho_0 = \rho_1 \leftrightarrow \rho_0 \subset \rho_1 \wedge \rho_0 \supset \rho_1$

▹ **Restriction:** $a_0\ \rho[\mathbb{A}_0]\ a_1 \leftrightarrow a_0 \in \mathbb{A}_0 \wedge a_1 \in \mathbb{A}_0 \wedge a_0\ \rho\ a_1$

▹ **Inversion:** $a_0\ \rho^-\ a_1 \leftrightarrow a_1\ \rho\ a_0$

▹ **Composition:** $a_0(\rho_0 \circ \rho_1)a_1 \leftrightarrow (\exists a_2)(a_0\ \rho_0\ a_2 \wedge a_2\ \rho_1\ a_1)$

▹ **Union:** $a_0(\cup\Gamma)a_1 \leftrightarrow (\exists\, \rho\in \Gamma)(a_0\ \rho\ a_1)$

▹ **Intersection:** $a_0(\cap\Gamma)a_1 \leftrightarrow (\forall\, \rho\in \Gamma)(a_0\ \rho\ a_1)$

▹ **Inversion Closure:** $\rho^\sim \underset{\text{df}}{=} \rho \cap \rho^-$

▹ **Transitive Closure:** ρ^+ is the least relation such that

$$a_0\ \rho\ a_1 \rightarrow a_0\ \rho^+\ a_1 \quad \text{and} \quad a_0\ \rho^+\ a_1 \wedge a_1\ \rho\ a_2 \rightarrow a_0\ \rho^+\ a_2$$

▹ **Transitive Reflexive Closure:** ρ^* is the least relation such that

$$a\ \rho^*\ a \quad \text{and} \quad a_0\ \rho^*\ a_1 \wedge a_1\ \rho\ a_2 \rightarrow a_0\ \rho^*\ a_2$$

▹ **Transitive Union:** $\uplus\Gamma \underset{\text{df}}{=} (\cup\Gamma)^*$

▹ **Properties and operations on sets of relations:** If θ is an operation on binary relations we write $\theta(\Gamma)$ for $\{\theta(\rho) \mid \rho \in \Gamma\}$ and if Θ is a property of binary relations we write $\Theta(\Gamma)$ for $(\forall\rho \in \Gamma)\Theta(\rho)$.

Chapter IV. Tree-structured computation

In this chapter we describe the world of tree-structured computation. The basic computation primitives are: getting a value from the computation environment; pfn generation; application of a data operation to a data sequence; application of a pfn to a sequence from the computation domain; conditional based on a test for the empty sequence; generation of the empty sequence; concatenation of sequences; and selection of elements from sequences. Computation is a process of generating computation trees.

In §1 the underlying algebraic structure for this fragment of *Rum* is described. In §2 the rules for computation are given. They are interpreted informally as rules for generating computation trees, then formally as defining the evaluation, subcomputation, and reduces-to relations on descriptions. Viewing computation rules as rules for generating computation trees provides information about the process of computation and about the structures generated in this process. The computation relations provide a basis for expressing and proving properties of computation. Additional relations useful for expressing properties of computation are also defined. Some basic facts about the computation relations are given, including a computation induction theorem and theorems expressing the sense in which the two interpretations of the computation rules are equivalent.

In the remainder of the chapter we begin the work in *Rum*, defining pfns and stating some simple properties. We focus mainly on *pure* pfns, which are the pfns common to all *Rum* worlds. In §3 notation and conventions for working in *Rum* are given. In §4 we build a small library of pfn definitions. We begin with a collection of *combinatory* pfns – pfns defined using only the abstraction and application computation primitives. These examples illustrate how the standard sorts of combinators are represented by pfns. They also illustrate additional properties that can be expressed in *Rum* and point out differences between *Rum* and traditional recursion and computation theories. The main result is the recursion theorem for *Rum* which gives a recursion pfn that computes computationally minimal fixed points. Using the recursion pfn and sequence primitives, pfns are defined that describe some standard computation schemes which are naturally formulated as recursion on sequences. In §5 pfns describing additional operations on S-expressions are defined. This provides concrete examples of computing in *Rum*, illustrates the use of pfns from the pfn library, and sets the stage for further work in the S-expression world. In §6 we show how streams can be represented in *Rum* and give examples of operations on streams and properties characterizing streams.

The point of these examples is to give the reader experience working in *Rum*, to develop some intuitions about *Rum* computation, and to give a sample of the variety things to be said about and done with pfns. More substantial examples are given in Appendix B.

IV.1. The objects of *Rum*

There are several *Rum* worlds including the world of tree-structured computation (t-*Rum*) and the world of sequential computation (s-*Rum*). The raw materials for a *Rum* world are a set of symbols $\mathbb{S}y$ and a data structure $\mathfrak{D}$. Data structures were discussed in §III.5. For the present, symbols are just atomic entities that serve as identifiers. We assume there are countably many symbols. The objects making up a *Rum* world are inductively generated uniformly from the given structures. The inductive definition has essentially the same form for each world, the differences being addition or omission of clauses in the definition. The sorts of objects making up a *Rum* world are summarized in Figure 9. In this section we discuss those objects needed for tree-structured computation. The remaining objects are used only in sequential computation and will be discussed in §V.1. To simplify notation, we fix certain variables to range over most sorts of objects. For example, $\mathfrak{f}$ ranges over $\mathbb{F}$, the set of forms. This in indicated in Figure 9 by $\mathfrak{f} \in \mathbb{F}$. A variable with a subscript ranges over the same sort as the unsubscripted variable. Thus $\mathfrak{f}_0$ and $\mathfrak{f}_{\mathrm{lhs}}$ also range over $\mathbb{F}$. In logical formulae, quantifiers are to be interpreted as quantifying over the range of the variables being quantified. Thus $(\forall\mathfrak{f})$ means "for all forms $\mathfrak{f}$".

Since constructions are uniformly parameterized by the data structure $\mathfrak{D}$ and the set of symbols $\mathbb{S}y$, we should refer to $Rum_{(\mathfrak{D},\mathbb{S}y)}$, $\mathbb{F}_{(\mathfrak{D},\mathbb{S}y)}$, etc. However, we will generally work with a fixed set of symbols and a fixed data structure and omit explicit mention of these parameters. In addition, the interpretation of domain symbols such as $\mathbb{F}$ and the range of the variable symbols of that sort depends on which world we are working in. To refer to a particular world we use a qualifier: t- for the world of tree-structured computation and s- for the world of sequential computation. For example elements of t-$\mathbb{F}$ are the forms of t-*Rum* and when working in t-*Rum* $(\forall\mathfrak{f})$ means for all forms in t-$\mathbb{F}$. Generally we work in a fixed world and the qualifier is omitted. In this chapter, since we are working in the world of tree-structured computation, *Rum* is interpreted as t-*Rum*, $\mathbb{F}$ is interpreted as t-$\mathbb{F}$, etc.

Name	Notation	Description
Data	$\mathbb{D}$	domain of the given data structure
	$d \in \mathbb{D}^*$	sequences of data
Operations	$o \in \mathbb{O}$	operations of the given data structure
Symbols	$s \in \mathbb{S}_y$	for naming values
Forms	$\mathfrak{f} \in \mathbb{F}$	symbolic descriptions of computation
Environments	$\xi \in \mathbb{E}$	finite maps that bind values to symbols
Dtrees	$\mathfrak{d} \in \mathbb{D}_t$	descriptions of particular computations
Pfns	$\varphi \in \mathbb{P}$	descriptions of partial functions
Computation Domain	$a \in \mathbb{V}$	data, data operations, pfns, continuations
Values	$u, v \in \mathbb{V}^*$	sequences from the computation domain
Stages	$\tau \in \mathbb{C}_s$	partial computation trees
Continuations	$\gamma \in \mathbb{C}_o$	descriptions of computation contexts
States	$\varsigma \in \mathbb{S}_t$	stages of sequential computation

Figure 9. Objects and notation

IV.1.1. Forms for tree-structured computation

Forms are the basic syntactic entities of *Rum*. They serve as symbolic descriptions of computation and also as terms in a language for expressing properties of the computation domain. Forms are freely generated over the set of symbols by the construction rules given in Figure 10. A form is either a symbol or constructed using λ, app, if, mt, cart, fst, rst. Each construction corresponds to a computation primitive, as indicated in Figure 10.

The construction rules are presented as expressions from which we read off such information as the official notation, the type, and names of components of the corresponding construction. The rule (mt) says that mt is a form. From the rule (app) we see that app($\mathfrak{f}_{\text{fun}}$,$\mathfrak{f}_{\text{arg}}$) is the form constructed by applying the app constructor operation to the forms $\mathfrak{f}_{\text{fun}}$ and $\mathfrak{f}_{\text{arg}}$. We also read from (app) that app $\in [\mathbb{F} \times \mathbb{F} \rightarrowtail \mathbb{F}]$, the first argument for app is the fun-component, and the second argument is the arg-component. A form constructed according to the rule (app) is called an app-form. From the rule (lam) have $\lambda \in [\mathbb{S}_y \times \mathbb{F} \rightarrowtail \mathbb{F}]$.

Two forms are equal exactly when they have the same construction. λ is a binding construct. Bound and free symbols in forms are determined in the usual manner, with free occurrences of s in $\mathfrak{f}$ bound by the outer λ-construction in $\lambda(s)\mathfrak{f}$.

Rule	Form	Computation Primitive
(sym)	s	naming values
(lam)	$\lambda(s_{\mathrm{arg}})\mathfrak{f}_{\mathrm{body}}$	pfn generation
(app)	$\mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}})$	application
(if)	$\mathsf{if}(\mathfrak{f}_{\mathrm{test}},\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}})$	conditional choice of subcomputation
(mt)	mt	empty sequence generation
(cart)	$\mathsf{cart}(\mathfrak{f}_{\mathrm{lhs}},\mathfrak{f}_{\mathrm{rhs}})$	sequence concatenation
(fst)	$\mathsf{fst}(\mathfrak{f}_{\mathrm{seq}})$	selection of first element of a sequence
(rst)	$\mathsf{rst}(\mathfrak{f}_{\mathrm{seq}})$	selection of remainder a sequence

Figure 10. Forms

frees$(\mathfrak{f})$ is the set of symbols occurring free in $\mathfrak{f}$ and $\mathfrak{f}|^{s}_{\mathfrak{f}_0}$ is the result of replacing free occurrences of s in $\mathfrak{f}$ by $\mathfrak{f}_0$ (with renaming of bound symbols in $\mathfrak{f}$ if necessary to avoid trapping of free symbols in $\mathfrak{f}_0$).

IV.1.2. Semantic domains for tree-structured computation

The remaining sorts of objects in t-*Rum* are dtrees, pfns, environments, the computation domain, and computation stages. They are semantic entities in the sense that they are used to provide interpretations of forms. Computation stages will be discussed in §2. The remaining objects are generated from forms and the given data and data operations by a mutual inductive definition. The domains generated are (isomorphic to) the minimal solutions to the following equations, modulo some additional rules for equality ($\overset{\delta}{=}$) which will be explained below.

(dtree) $\mathbb{D}t \sim (\mathbb{F} \times \mathbb{E})_{\overset{\delta}{=}} \oplus (\mathbb{O} \times \mathbb{D}^*)$

(pfn) $\mathbb{P} \sim (\mathbb{S}y \times \mathbb{F} \times \mathbb{E})_{\overset{\delta}{=}}$

(environment) $\mathbb{E} \sim [\mathbb{S}y \ast\!\succ \mathbf{V}^*]_{\alpha}$

(computation domain) $\mathbf{V} \sim \mathbb{D} \oplus \mathbb{O} \oplus \mathbb{P}$

- **Dtrees.** The dtree equation

$$\mathbb{D}t = (\mathbb{F} \times \mathbb{E})_{\overset{\delta}{=}} \oplus (\mathbb{O} \times \mathbb{D}^*)$$

says that a dtree is constructed either from a data operation and a data sequence, or from a form and an environment.

Data application dtrees. We call a dtree constructed from a data operation and a data sequence a data application dtree. $o'd$ is the data application dtree constructed from the data operation o and the data sequence d.

Closure dtrees. We call a dtree constructed from a form and an environment a closure dtree. The closure of the form $\mathfrak{f}$ in the environment ξ is written $\langle \mathfrak{f} \mid \xi \rangle$. The closure operation constructs a finite tree structure in which free symbols of the form are replaced by value nodes containing the value bound to the symbol in the environment (the empty sequence, if the symbol is not explicitly bound in the environment). λ-bound symbols are replaced by binding arrows pointing to the binding node. Thus dtrees contain no symbols. The computation environment of a dtree is the set of value nodes of the dtree, each such node being associated with the value occurring at that node.

Closure dtrees are generated freely from value nodes by constructions corresponding to the form constructions and are classified according to these constructions (see Appendix A). For non-binding constructions, the dtree components are the corresponding form components closed in the same environment. For example, $\langle \mathsf{app}(\mathfrak{f}_{\mathrm{fun}}, \mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle$ is an app-dtree. It has fun-component $\langle \mathfrak{f}_{\mathrm{fun}} \mid \xi \rangle$ and arg-component $\langle \mathfrak{f}_{\mathrm{arg}} \mid \xi \rangle$.

Dtree equality. Two dtrees are equal only if they are both data application dtrees or both closure dtrees. Two data application dtrees are equal iff the corresponding components are equal. The closures of different form environment pairs may be equal dtrees. This corresponds to the fact that two form environment pairs may be different, but still describe the same computation. Two closure dtrees are equal just when they describe the same computation. The main point of introducing the closure construction is to capture this notion of equality. For example, form environment pairs that differ by renaming of environment bound symbols (i) and (ii), renaming of λ- bound symbols (iii), or modifying bindings of symbols not free in the form (iv) describe the same computation.

(i) $\langle \mathsf{x} \mid \xi \rangle = \langle \mathsf{y} \mid \xi \rangle \quad \text{if} \quad \xi(\mathsf{x}) = \xi(\mathsf{y})$

(ii) $\langle \mathsf{cart}(\mathsf{x},\mathsf{x}) \mid \xi \rangle = \langle \mathsf{cart}(\mathsf{x},\mathsf{y}) \mid \xi \rangle = \langle \mathsf{cart}(\mathsf{z},\mathsf{z}) \mid \xi \rangle \quad \text{if} \quad \xi(\mathsf{x}) = \xi(\mathsf{y}) = \xi(\mathsf{z})$

(iii) $\langle \lambda(\mathsf{x})\mathsf{app}(\mathsf{f},\mathsf{x}) \mid \xi \rangle = \langle \lambda(\mathsf{y})\mathsf{app}(\mathsf{f},\mathsf{y}) \mid \xi \rangle$

(iv) $\langle \mathsf{x} \mid \xi \rangle = \langle \mathsf{x} \mid \xi\{\mathsf{y} \leftarrow v\} \rangle$

The equivalence $\overset{\delta}{=}$ on form-environment pairs is generated by renaming of λ and environment bound symbols and by modifying bindings of symbols not free in the form and corresponds exactly to closure dtree equality. Thus equality of dtrees given as closures can be determined with out knowing the details of dtree structure. (This equivalence is defined precisely in Appendix A.)

Since only the values of symbols free in $\mathfrak{f}$ are relevant in a closure dtree we may replace the environment component by a list of bindings for the free symbols of $\mathfrak{f}$. If there are no free symbols we may forget the environment entirely. For example, we write $\langle \mathsf{x} \mid \mathsf{x} \leftarrow \xi(\mathsf{x})\rangle$ for $\langle \mathsf{x} \mid \xi\rangle$ and $\lambda(\mathsf{x})\mathsf{x}$ for $\langle \lambda(\mathsf{x})\mathsf{x} \mid \rangle$.

- **Pfns.** The equation for pfns

$$\mathbb{P} = (\mathbb{S}_y \times \mathbb{F} \times \mathbb{E})_{\underline{\underline{\delta}}}$$

says that each pfn is constructed from a symbol, a form and an environment. We write $\langle \lambda(s_{\text{arg}})\mathfrak{f}_{\text{body}} \mid \xi\rangle$ for the pfn constructed from s_{arg}, $\mathfrak{f}_{\text{body}}$ and ξ. s_{arg} is the pfn argument symbol, $\mathfrak{f}_{\text{body}}$ is the pfn body, and ξ is the pfn environment. We call this the abstraction of $\mathfrak{f}_{\text{body}}$ with respect to s_{arg} in ξ. Structurally, pfns are just λ-dtrees, thus pfn equality is dtree equality restricted to pfns. Note that, as for dtrees, the same pfn may be given by many different symbol, form, environment triples.

We extend the application notation to pfns by defining

$$\langle \lambda(s)\mathfrak{f} \mid \xi\rangle\,{}'\,v \underset{\text{df}}{=} \langle \mathfrak{f} \mid \xi\{s \leftarrow v\}\rangle$$

We say that an expression $v_{\mathfrak{f}}\,{}'\,v_{\text{a}}$ is *well-formed* if $v_{\mathfrak{f}}$ is a data operation and v_{a} is a data sequence or if $v_{\mathfrak{f}}$ is a pfn.

Although structurally pfns and dtrees are the same, they play different roles as descriptions of computation. The dtree $\langle \lambda(s_{\text{arg}})\mathfrak{f}_{\text{body}} \mid \xi\rangle$ describes the primitive computation which returns the pfn $\langle \lambda(s_{\text{arg}})\mathfrak{f}_{\text{body}} \mid \xi\rangle$ as a value. The pfn $\langle \lambda(s_{\text{arg}})\mathfrak{f}_{\text{body}} \mid \xi\rangle$ describes a family of computations $\langle \lambda(s_{\text{arg}})\mathfrak{f}_{\text{body}} \mid \xi\rangle\,{}'\,v$. In general the context determines which interpretation is intended. The need for a formal distinction will become clearer when we discuss machine structures and morphisms in Chapter VII.

- **Environments.** The equation for environments

$$\mathbb{E} = [\mathbb{S}_y \ast\!\succ \mathbb{V}^*]_{\square}$$

says that environments are the set of finite maps from symbols to values with default the empty sequence. Thus environments are generated according to the rules for finite maps given in §III.3. Note that the empty environment $\{\}$ maps each symbol to the empty sequence.

• **The computation domain.** The equation for the computation domain

$$\mathbf{V} = \mathbb{D} \oplus \mathbb{O} \oplus \mathbb{P}$$

says that the computation domain $\mathbf{V}$ consists disjointly of data, data operations, and pfns. Finite sequences from the computation domain (elements of $\mathbf{V}^*$) are called values. They are generated according to the rules for finite sequences given in §III.2.

IV.2. Rules for tree-structured computation

IV.2.1. Computation primitives

Each sort of form corresponds to a basic computation primitive. The computation described by a form relative to a computation environment applies the corresponding primitive using results of subcomputations as parameters. A symbol s describes looking up its value in the computation environment. A lam-form $\lambda(s_{\mathrm{arg}})\mathfrak{f}_{\mathrm{body}}$ describes generation of the pfn given by the symbol s_{arg}, the form $\mathfrak{f}_{\mathrm{body}}$ and the computation environment. An app-form $\mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}})$ describes the application of the result of the fun-subcomputation. to the result of the arg-subcomputation. If the fun-subcomputation, described by $\mathfrak{f}_{\mathrm{fun}}$, returns value v_{fun} and the arg-subcomputation, described by $\mathfrak{f}_{\mathrm{arg}}$, returns value v_{arg} then the computation proceeds by carrying out the computation described by $v_{\mathrm{fun}}{}'v_{\mathrm{arg}}$ (if well-formed), returning the value returned by this subcomputation, (if any). An if-form $\mathsf{if}(\mathfrak{f}_{\mathrm{test}},\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}})$ describes a test $\mathfrak{f}_{\mathrm{test}}$ and two branch subcomputations. The test-subcomputation is carried out first. If the value returned by the test-subcomputation is a non-empty sequence (resp. the empty sequence) then computation proceeds by carrying out the then- (resp. else-) subcomputation, returning the value returned by this subcomputation, (if any). The form mt generates the empty sequence. A cart-form $\mathsf{cart}(\mathfrak{f}_{\mathrm{lhs}},\mathfrak{f}_{\mathrm{rhs}})$ describes concatenation of results of the lhs- and rhs-subcomputations. (Think of carrying around a collection of things lined up in a "cart".) A fst-form $\mathsf{fst}(\mathfrak{f}_{\mathrm{seq}})$ describes selecting the first element of the sequence returned by $\mathfrak{f}_{\mathrm{seq}}$ and a rst-form $\mathsf{rst}(\mathfrak{f}_{\mathrm{seq}})$ describes selecting the rest of the sequence returned by $\mathfrak{f}_{\mathrm{seq}}$.

IV.2.2. Generation of computation trees

Figure 11 gives the rules for tree-structured computation. These rules make precise the informal description of computation given above. The rules are presented as logical formulae and can be interpreted as clauses of an inductive definition of the relation symbols $\hookrightarrow$, $\prec_{\iota}$, and $\succ\!\!\succ_{\iota}$. This interpretation will be elaborated

Return Rules / The Evaluation Relation: $\hookrightarrow \in [\mathbb{D}t \rightsquigarrow \mathbf{V}^*]$

(sym) $\langle s \mid \xi \rangle \hookrightarrow \xi(s)$

(lam) $\langle \lambda(s)\mathfrak{f} \mid \xi \rangle \hookrightarrow \langle \lambda(s)\mathfrak{f} \mid \xi \rangle$

(dapp) $o\,'\,d \hookrightarrow o(d)$

(mt) $\mathsf{mt} \hookrightarrow \square$

(cart) $\langle \mathfrak{f}_{\mathbf{lhs}} \mid \xi \rangle \hookrightarrow v_{\mathbf{lhs}} \wedge \langle \mathfrak{f}_{\mathbf{rhs}} \mid \xi \rangle \hookrightarrow v_{\mathbf{rhs}} \rightarrow \langle \mathsf{cart}(\mathfrak{f}_{\mathbf{lhs}}, \mathfrak{f}_{\mathbf{rhs}}) \mid \xi \rangle \hookrightarrow [v_{\mathbf{lhs}}, v_{\mathbf{rhs}}]$

(fst) $\langle \mathfrak{f}_{\mathbf{seq}} \mid \xi \rangle \hookrightarrow v_{\mathbf{seq}} \rightarrow \langle \mathsf{fst}(\mathfrak{f}_{\mathbf{seq}}) \mid \xi \rangle \hookrightarrow 1^{\mathbf{st}}\,(v_{\mathbf{seq}})$

(rst) $\langle \mathfrak{f}_{\mathbf{seq}} \mid \xi \rangle \hookrightarrow v_{\mathbf{seq}} \rightarrow \langle \mathsf{rst}(\mathfrak{f}_{\mathbf{seq}}) \mid \xi \rangle \hookrightarrow \mathrm{r}^{\mathbf{st}}\,(v_{\mathbf{seq}})$

(ret) $\mathfrak{d} \succ\!\!\succ_\iota \mathfrak{d}_0 \wedge \mathfrak{d}_0 \hookrightarrow v \rightarrow \mathfrak{d} \hookrightarrow v$

Reduces-to Rules: $\succ\!\!\succ_\iota \in [\mathbb{D}t \rightsquigarrow \mathbb{D}t]$

(app) $\langle \mathfrak{f}_{\mathbf{fun}} \mid \xi \rangle \hookrightarrow v_{\mathbf{fun}} \wedge \langle \mathfrak{f}_{\mathbf{arg}} \mid \xi \rangle \hookrightarrow v_{\mathbf{arg}} \rightarrow \langle \mathsf{app}(\mathfrak{f}_{\mathbf{fun}}, \mathfrak{f}_{\mathbf{arg}}) \mid \xi \rangle \succ\!\!\succ_\iota v_{\mathbf{fun}}\,'\,v_{\mathbf{arg}}{}^{\dagger}$

(if) $\langle \mathfrak{f}_{\mathbf{test}} \mid \xi \rangle \hookrightarrow u \rightarrow \langle \mathsf{if}(\mathfrak{f}_{\mathbf{test}}, \mathfrak{f}_{\mathbf{then}}, \mathfrak{f}_{\mathbf{else}}) \mid \xi \rangle \succ\!\!\succ_\iota \begin{cases} \langle \mathfrak{f}_{\mathbf{then}} \mid \xi \rangle & \text{if } u \neq \square \\ \langle \mathfrak{f}_{\mathbf{else}} \mid \xi \rangle & \text{if } u = \square \end{cases}$

Begin Rules / The Subcomputation Relation: $\prec_\iota \subset [\mathbb{D}t \times \mathbb{D}t]$

(if.test) $\langle \mathfrak{f}_{\mathbf{test}} \mid \xi \rangle \prec_\iota \langle \mathsf{if}(\mathfrak{f}_{\mathbf{test}}, \mathfrak{f}_{\mathbf{then}}, \mathfrak{f}_{\mathbf{else}}) \mid \xi \rangle$

(app.fun) $\langle \mathfrak{f}_{\mathbf{fun}} \mid \xi \rangle \prec_\iota \langle \mathsf{app}(\mathfrak{f}_{\mathbf{fun}}, \mathfrak{f}_{\mathbf{arg}}) \mid \xi \rangle$

(app.arg) $\langle \mathfrak{f}_{\mathbf{arg}} \mid \xi \rangle \prec_\iota \langle \mathsf{app}(\mathfrak{f}_{\mathbf{fun}}, \mathfrak{f}_{\mathbf{arg}}) \mid \xi \rangle$

(cart.lhs) $\langle \mathfrak{f}_{\mathbf{lhs}} \mid \xi \rangle \prec_\iota \langle \mathsf{cart}(\mathfrak{f}_{\mathbf{lhs}}, \mathfrak{f}_{\mathbf{rhs}}) \mid \xi \rangle$

(cart.rhs) $\langle \mathfrak{f}_{\mathbf{rhs}} \mid \xi \rangle \prec_\iota \langle \mathsf{cart}(\mathfrak{f}_{\mathbf{lhs}}, \mathfrak{f}_{\mathbf{rhs}}) \mid \xi \rangle$

(fst.seq) $\langle \mathfrak{f}_{\mathbf{seq}} \mid \xi \rangle \prec_\iota \langle \mathsf{fst}(\mathfrak{f}_{\mathbf{seq}}) \mid \xi \rangle$

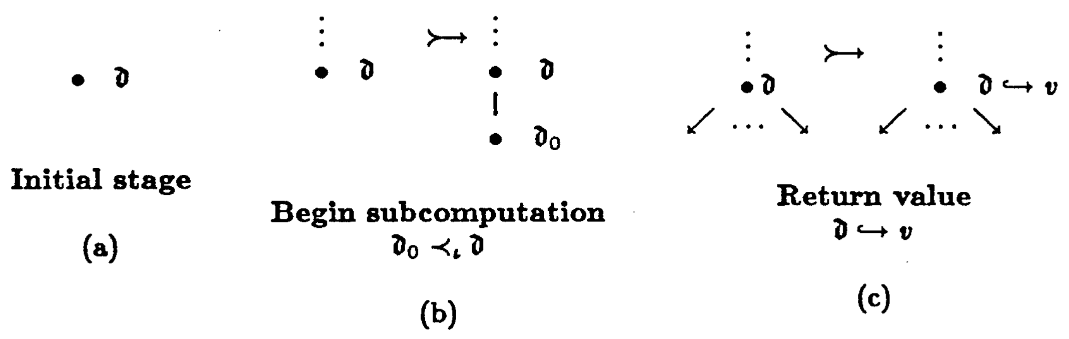
(rst.seq) $\langle \mathfrak{f}_{\mathbf{seq}} \mid \xi \rangle \prec_\iota \langle \mathsf{rst}(\mathfrak{f}_{\mathbf{seq}}) \mid \xi \rangle$

(redt.beg) $\mathfrak{d} \succ\!\!\succ_\iota \mathfrak{d}_0 \rightarrow \mathfrak{d}_0 \prec_\iota \mathfrak{d}$

[†] if $v_{\mathbf{fun}}\,'\,v_{\mathbf{arg}}$ is well-formed – $v_{\mathbf{fun}} \in \mathbb{O} \wedge v_{\mathbf{arg}} \in \mathfrak{D}^*$ or $v_{\mathbf{fun}} \in \mathbb{P}$.

Figure 11. Rules for tree-structured computation

below. First we explain briefly the interpretation of the computation rules as rules for generating computation trees and give pictures for two simple examples. Our intent is simply to provide geometric intuition for the structure computation trees and for intensional properties of computations carried out by generating computation trees. The mathematical structures representing computations trees and operations on these structures are defined in Appendix A.



Initial stage

(a)

Begin subcomputation
$\mathfrak{d}_0 \prec_\iota \mathfrak{d}$

(b)

Return value
$\mathfrak{d} \hookrightarrow v$

(c)

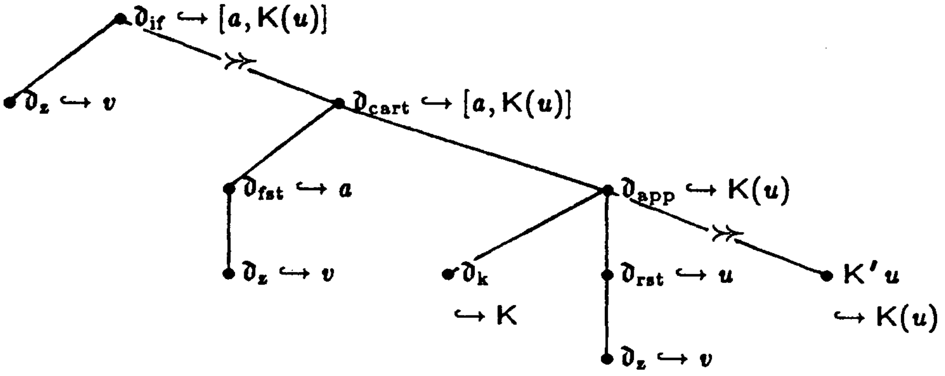
Figure 12. Generating computation trees

Computation trees are generated by stepwise extensions of partial computation trees, called computation stages. A computation stage is a finite tree structure. Each node is labeled by the dtree describing the subtree below that node. The root of a completed subtree is also labeled by the value returned by that subcomputation. The initial stage of the computation described by $\mathfrak{d}$ is a single node with dtree label, $\mathfrak{d}$ – Figure 12 (a). There are two sorts of steps – begin a subcomputation (apply a begin rule) and return a value (apply a return rule).[1] A begin rule $\mathfrak{d}_0 \prec_\iota \mathfrak{d}$ applies at a node labeled $\mathfrak{d}$ if the rule has not been used at that node. The subcomputation is begun by adding a new successor node below the given node. The new node is labeled by the dtree $\mathfrak{d}_0$ – Figure 12 (b). A return rule with conclusion $\mathfrak{d} \hookrightarrow v$ applies at a node labeled $\mathfrak{d}$ if the immediate subcomputations mentioned in the rule premiss have returned values matching those in the premiss. A value is returned at a node by adding it as the value label of that node – Figure 12 (c). Each reduces-to rule combines with the rule (redt.beg) to

[1] This is analogous to the classification of instructions given by Wegner [1972] (see §I.4) for transformations on VDL computation states. Begin rules correspond to macro instructions and return rules correspond to value instructions.

give a begin rule and with (redt.ret) to give a return rule (transitivity of implication). The arc leading to a reduced-to subcomputation may be labeled by with the reduces-to sign $\succ\!\!\succ$ for emphasis.

The computation tree for a dtree $\mathfrak{d}$ is the limit of the stages reachable from the initial stage by extensions according to the computation rules. At any stage where more than one rule applies, the corresponding extensions may be made in any order with the same result. Thus the limit is well defined. Since the process of computation may not terminate in finitely many steps, computation trees may be infinite, although stages and descriptions are finite. The computation tree described by a dtree $\mathfrak{d}$ returns a value v if the root of the computation tree for $\mathfrak{d}$ has value label v. A computation tree which returns a value is finite and all nodes have a value label.



$\mathfrak{d}_{\mathrm{if}} \hookrightarrow [a, \mathsf{K}(u)]$
$\succ\!\!\succ$
$\mathfrak{d}_{\mathrm{z}} \hookrightarrow v$
$\mathfrak{d}_{\mathrm{cart}} \hookrightarrow [a, \mathsf{K}(u)]$
$\mathfrak{d}_{\mathrm{fst}} \hookrightarrow a$
$\mathfrak{d}_{\mathrm{app}} \hookrightarrow \mathsf{K}(u)$
$\succ\!\!\succ$
$\mathfrak{d}_{\mathrm{z}} \hookrightarrow v$
$\mathfrak{d}_{\mathrm{k}} \hookrightarrow \mathsf{K}$
$\mathfrak{d}_{\mathrm{rst}} \hookrightarrow u$
$\mathsf{K}'u \hookrightarrow \mathsf{K}(u)$
$\mathfrak{d}_{\mathrm{z}} \hookrightarrow v$

where

$$\mathfrak{d}_{\mathrm{if}} = \langle \mathsf{if(z,cart(fst(z),app(\lambda(x)\lambda(y)x,rst(z))),mt)} \mid \mathsf{z} \leftarrow v\rangle$$

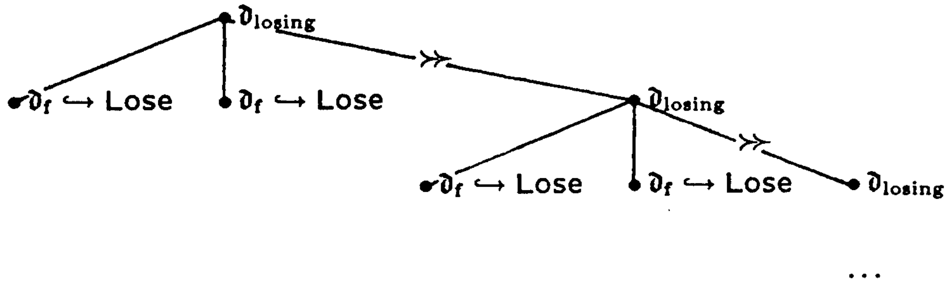
$$\mathfrak{d}_{\mathrm{cart}} = \langle \mathsf{cart(fst(z),app(\lambda(x)\lambda(y)x,rst(z)))} \mid \mathsf{z} \leftarrow v\rangle$$

$$\mathfrak{d}_{\mathrm{fst}} = \langle \mathsf{fst(z)} \mid \mathsf{z} \leftarrow v\rangle \qquad \mathfrak{d}_{\mathrm{app}} = \langle \mathsf{app(\lambda(x)\lambda(y)x,rst(z))} \mid \mathsf{z} \leftarrow v\rangle$$

$$\mathfrak{d}_{\mathrm{z}} = \langle \mathsf{z} \mid \mathsf{z} \leftarrow v\rangle \qquad \mathfrak{d}_{\mathrm{rst}} = \langle \mathsf{rst(z)} \mid \mathsf{z} \leftarrow v\rangle \qquad v = [a, u]$$

$$\mathfrak{d}_{\mathrm{k}} = \lambda(\mathsf{x})\lambda(\mathsf{y})\mathsf{x} \qquad \mathsf{K} = \lambda(\mathsf{x})\lambda(\mathsf{y})\mathsf{x} \qquad \mathsf{K}(u) = \langle \lambda(\mathsf{y})\mathsf{x} \mid \mathsf{x} \leftarrow u\rangle$$

Figure 13. Computation tree for $\mathfrak{d}_{\mathrm{if}}$


$\mathfrak{d}_{\mathrm{losing}}$
$\mathfrak{d}_{\mathrm{f}} \hookrightarrow \mathsf{Lose}$
$\mathfrak{d}_{\mathrm{f}} \hookrightarrow \mathsf{Lose}$
$\mathfrak{d}_{\mathrm{losing}}$
$\mathfrak{d}_{\mathrm{f}} \hookrightarrow \mathsf{Lose}$
$\mathfrak{d}_{\mathrm{f}} \hookrightarrow \mathsf{Lose}$
$\mathfrak{d}_{\mathrm{losing}}$
...

where $\mathfrak{d}_{\mathrm{losing}} = \mathsf{Lose}'\mathsf{Lose}$ and $\mathfrak{d}_{\mathrm{f}} = \langle \mathsf{f} \mid \mathsf{f} \leftarrowtail \mathsf{Lose} \rangle$

Figure 14. A losing computation

Example: A finite computation tree. Figure 13 shows the computation tree described by the dtree $\mathfrak{d}_{\mathrm{if}}$.

$$\mathfrak{d}_{\mathrm{if}} = \langle \mathsf{if(z,cart(fst(z),app(}\lambda\mathsf{(x)}\lambda\mathsf{(y)x,rst(z))),mt)} \mid \mathsf{z} \leftarrowtail v \rangle$$

The description reads: if z is a non-empty sequence then return the sequence containing the first element of z and the result of applying $\lambda(\mathsf{x})\lambda(\mathsf{y})\mathsf{x}$ to the rest of z, otherwise return the empty sequence. We use K to denote the pfn $\lambda(\mathsf{x})\lambda(\mathsf{y})\mathsf{x}$ in analogy to the "K" combinator. $\mathsf{K}(u)$ is the pfn $\langle \lambda(\mathsf{y})\mathsf{x} \mid \mathsf{x} \leftarrowtail u \rangle$, the result of applying K to u. From the computation tree for $\mathfrak{d}_{\mathrm{if}}$ we can read off facts about the computation structure such as

(i) the value returned by $\mathfrak{d}_{\mathrm{if}}$ is $[a, \mathsf{K}(u)]$

(ii) $\mathfrak{d}_{\mathrm{fst}}$ is an immediate subcomputation of $\mathfrak{d}_{\mathrm{cart}}$

(iii) computation of $\mathfrak{d}_{\mathrm{app}}$ reduces-to computation of $\mathsf{K}'u$

We can also read off other information about the computation such as the rule (sym) was applied three times, the rule (lam) two times, the rule (cart) one time, there was one pfn application, and the depth of the tree is four.

Example: An infinite computation tree. As a second example we describe what is perhaps the simplest non-terminating computation. Let $\mathsf{Lose} = \lambda(\mathsf{f})\mathsf{app(f,f)}$. Then Lose is a pfn that applies any argument to itself. Applying Lose to itself we obtain the dtree $\mathfrak{d}_{\mathrm{losing}}$

$$\mathfrak{d}_{\mathrm{losing}} = \mathsf{Lose}'\mathsf{Lose} = \langle \mathsf{app(f,f)} \mid \mathsf{f} \leftarrowtail \mathsf{Lose} \rangle$$

which reads "apply f to f where f is bound to Lose." The computation tree described by $\mathfrak{d}_{\mathrm{losing}}$ is shown in Figure 14.

IV.2.3. Computation relations

We now interpret the computation rules as clauses of an inductive definition. Thus $\hookrightarrow$, $\prec_\iota$, and $\succ\!\!\succ_\iota$ are the least relations on $\mathbb{D}t \times \mathbf{V}^*$, $\mathbb{D}t \times \mathbb{D}t$, and $\mathbb{D}t \times \mathbb{D}t$, respectively, satisfying the closure conditions (implications) expressed by the computation rules. The following basic facts follow easily.

- Evaluation and immediately reduces-to are partial functions.

(eval.fun) $\eth \hookrightarrow v_0 \wedge \eth \hookrightarrow v_1 \;\rightarrow\; v_0 = v_1$

(redt.fun) $\eth_0 \succ\!\!\succ_\iota \eth_1 \wedge \eth_0 \succ\!\!\succ_\iota \eth_2 \;\rightarrow\; \eth_1 = \eth_2$

Remark. The conditional if tests for equality to the empty sequence. Equality between arbitrary elements of the computation domain is not computable in *Rum*. In §5 it is shown that equality on the S-expression domain is computable. In general, equality between elements of the data domain may or may not be computable, depending on the given data operations.

IV.2.3.1. Computation trees vs. computation relations

The two interpretations given for the computation rules are essentially the same. By expressing the rules for generating computation trees in terms of operations on tree structures it is easy to show the following. (See Appendix A for a precise formulation.)

- Evaluation ($\hookrightarrow$) is the relation on dtrees and values that gives the value returned by the described computation (if any). $\eth \hookrightarrow v$ iff the computation described by $\eth$ returns the value v.

- Immediate subcomputation ($\prec_\iota$) is the relation on dtrees that corresponds to the immediate subtree relation on computation trees. $\eth_0 \prec_\iota \eth$ iff the computation tree described by $\eth$ has an immediate successor node with dtree label $\eth_0$.

- Immediately reduces-to ($\succ\!\!\succ_\iota$) is a subrelation of immediate subcomputation. $\eth \succ\!\!\succ_\iota \eth_1$ when $\eth$ is an if- dtree and $\eth_1$ is the branch determined by the result of the test-subcomputation, or when $\eth$ is an app-dtree and $\eth_1$ is the application of the results of the fun- and arg-subcomputations.

Using the correspondence between computation trees and computation relations we can also read off instances of computation relations from a computation tree. For example from the tree for $\eth_{\mathrm{if}}$ we have

(i) $\eth_{\mathrm{if}} \hookrightarrow [a, \mathsf{K}(u)]$

(ii) $\eth_{\mathrm{fst}} \prec_\iota \eth_{\mathrm{cart}}$

(iii) $\eth_{\mathrm{app}} \succ\!\!\succ_\iota \mathsf{K}'u$

IV.2.3.2. More computation relations and facts

Some additional relations useful in formulating properties of computation are defined below.

▷ **Subcomputation.** The *subcomputation* relation ($\prec$) is the transitive closure of the immediate subcomputation relation.

▷ **Reduces-to.** The *reduces-to* relation ($\gg\!\!-$) is the transitive closure of the immediately reduces-to relation.

▷ **Definedness.** A dtree is *defined* iff the computation described by that dtree returns a value.

$$\Downarrow\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\exists v)\mathfrak{d} \hookrightarrow v$$

▷ **Computation tree equivalence.** Two dtrees are *computation tree equivalent* (abbreviated ctree equivalent) iff both are undefined or both are defined with the same value.[2]

$$\mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} (\forall v)(\mathfrak{d}_0 \hookrightarrow v \leftrightarrow \mathfrak{d}_1 \hookrightarrow v)$$

The following are some additional useful facts about tree-structured computation. They follow easily from the definitions.

■ If a computation returns a value then all subcomputations return values.

(def.subc) $$\Downarrow\mathfrak{d}_1 \wedge \mathfrak{d}_0 \prec \mathfrak{d}_1 \rightarrow \Downarrow\mathfrak{d}_0$$

■ If $\mathfrak{d}_0$ reduces-to $\mathfrak{d}_1$ then $\mathfrak{d}_0$ is weakly equal to $\mathfrak{d}_1$

(redt.ctree.eq) $$\mathfrak{d}_0 \gg\!\!- \mathfrak{d}_1 \rightarrow \mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1$$

■ Completed computation trees are well-founded. Thus for any dtree property Q we have the *computation induction* formula

(CI) $$(\forall\mathfrak{d})((\forall\mathfrak{d}_0 \prec \mathfrak{d})Q(\mathfrak{d}_0) \rightarrow Q(\mathfrak{d})) \rightarrow (\forall\mathfrak{d})(\Downarrow\mathfrak{d} \rightarrow Q(\mathfrak{d}))$$

■ A simple consequence of computation induction is that a dtree which is a subcomputation of itself is not defined.

(loop) $$\mathfrak{d} \prec \mathfrak{d} \rightarrow \neg\Downarrow\mathfrak{d}$$

An immediate corollary is $\neg\Downarrow$(Lose'Lose) which was asserted informally in the example of a "losing" computation.

[2] Computation tree equivalence is analogous to Kleene's *complete equality,* $\cong$ (Kleene [1952]).

IV.3. Notation and conventions

IV.3.1. The global context

While working in *Rum*, we imagine that we are working in a *global context*. This context determines a set of *global symbols* $\sigma_\star$ and a *global environment* $\xi_\star$ assigning values to the global symbols. We say an environment ξ is compatible with the global environment if it agrees with the global environment on global symbols – $(\forall s \in \sigma_\star)(\xi(s) = \xi_\star(s))$. A symbol definition adds a new symbol to the set of global symbols and extends the global environment by binding the defining value to the defined symbol. For example

$\triangleright$ $\quad \mathsf{I} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x})\mathsf{x}$

defines I to be the identity pfn, $\lambda(\mathsf{x})\mathsf{x}$. In general, symbol definitions have the format

$\triangleright$ $\quad < \mathsf{symbol} > \overset{\mathrm{df}}{\longleftrightarrow} < \mathsf{form} >$

where $< \mathsf{symbol} >$ is the symbol whose value is being defined, and $< \mathsf{form} >$ is a form that has a value in the global environment. The definition extends the global environment by binding the value of $< \mathsf{form} >$ (in $\xi_\star$) to $< \mathsf{symbol} >$. To help keep track of global symbols, we use identifiers such as I, K, and Rec, beginning with upper-case letters for global symbols. To simplify notation, we use a global symbol constant in a context where a value is expected to denote the value of that symbol in the global environment. For example we write $\mathsf{I}'v \hookrightarrow v$ rather than $\xi_\star(\mathsf{I})'v \hookrightarrow v$.

IV.3.2. Satisfaction in the global context

Many of the properties of computation that we study are properties of tuples of dtrees $(\langle \mathfrak{f}_1 \mid \xi\rangle, \ldots, \langle \mathfrak{f}_n \mid \xi\rangle)$ which hold for all environments ξ compatible with the global environment. For example $\langle \mathsf{I}(\mathsf{x}) \mid \xi\rangle \succ\!\!\succ \langle \mathsf{x} \mid \xi\rangle$ is such a property. To express such properties naturally, we define a notion of satisfaction which allows us to interpret dtree properties as properties of forms.

$\triangleright$ **Satisfaction by the global context.** For any n-ary dtree relation Θ, the global context satisfies $\Theta(\mathfrak{f}_1, \ldots, \mathfrak{f}_n)$ (written $\models_\star \Theta(\mathfrak{f}_1, \ldots, \mathfrak{f}_n)$) iff Θ holds for each tuple of closures with respect to an environment extending the the global environment.

$$\models_\star \Theta(\mathfrak{f}_1, \ldots, \mathfrak{f}_n) \underset{\mathrm{df}}{\equiv} (\forall \xi)(\forall s \in \sigma_\star)(\xi(s) = \xi_\star(s) \rightarrow \Theta(\langle \mathfrak{f}_1 \mid \xi\rangle, \ldots, \langle \mathfrak{f}_n \mid \xi\rangle))$$

Recall that $\sigma_\star$ is the set of global symbols and $\xi_\star$ is the global environment. We include as dtree properties logical formulae built from the basic dtree relations having only dtree variables free. We will generally omit the sign $\models_\star$ and write $\Theta(\mathfrak{f}_1,\ldots,\mathfrak{f}_n)$ for $\models_\star\ \Theta(\mathfrak{f}_1,\ldots,\mathfrak{f}_n)$. For example, the basic facts about subcomputation and reduces-to can be expressed by

(def.subc) $\Downarrow\mathfrak{f}_1 \wedge \mathfrak{f}_0 \prec \mathfrak{f}_1 \rightarrow \Downarrow\mathfrak{f}_0$

(redt.subc) $\mathfrak{f}_0 \gg \mathfrak{f}_1 \rightarrow \mathfrak{f}_0 \prec \mathfrak{f}_1$

(redt.ctree.eq) $\mathfrak{f}_0 \gg \mathfrak{f}_1 \rightarrow \mathfrak{f}_0 \rightleftharpoons \mathfrak{f}_1$

By our convention partitioning symbols into global and non-global symbols we avoid the possibility that at some point in the development the global context satisfies a given formula and at some other point the global context does not satisfy this formula.

IV.3.3. Notations for sequences and multiple arguments

Cart-forms may be written with any number of arguments using the usual conventions for associative binary operations.

$$\mathsf{cart}() \underset{\mathrm{df}}{=} \mathsf{mt}, \qquad \mathsf{cart}(\mathfrak{f}) \underset{\mathrm{df}}{=} \mathfrak{f}, \quad \text{and} \quad \mathsf{cart}(\mathfrak{f}_1,\mathfrak{f}_2 \ldots ,\mathfrak{f}_n) \underset{\mathrm{df}}{=} \mathsf{cart}(\mathfrak{f}_1,\mathsf{cart}(\mathfrak{f}_2 \ldots ,\mathfrak{f}_n))$$

and we write $[\mathfrak{f}_1, \ldots ,\mathfrak{f}_n]$ for $\mathsf{cart}(\mathfrak{f}_1, \ldots ,\mathfrak{f}_n)$.

There are two natural notions of multiple argument abstraction and application. The first is analogous to the usual extension of lambda notation by *currying.*

$$\begin{aligned} \lambda(s_1, \ldots ,s_n)\mathfrak{f} &\underset{\mathrm{df}}{=} \lambda(s_1) \ldots \lambda(s_n)\mathfrak{f} \\ \mathfrak{f}_0(\mathfrak{f}_1, \ldots ,\mathfrak{f}_n) &\underset{\mathrm{df}}{=} \mathsf{app}(\ldots \mathsf{app}(\mathfrak{f}_0,\mathfrak{f}_1), \ldots \mathfrak{f}_n) \end{aligned}$$

The second means of expressing multiple arguments is to match a sequence of symbols to elements of an argument sequence. For example, $\lambda[s_1,s_2,s_3]\mathfrak{f}$ binds the first element of an argument to s_1, the second element to s_2 and the second remainder to s_3. In general,

$$\begin{aligned} \lambda[s]\mathfrak{f} &\underset{\mathrm{df}}{=} \lambda(s)\mathfrak{f} \\ \lambda[s_1,s_2, \ldots ,s_n]\mathfrak{f} &\underset{\mathrm{df}}{=} \lambda(s_1)(\{\lambda(s_1)\lambda[s_2, \ldots ,s_n]\mathfrak{f}\}(\mathsf{fst}(s_1),\mathsf{rst}(s_1))). \end{aligned}$$

Combining the two modes of expressing multiple arguments, we may write expressions such as $\lambda(\ldots,b_i,\ldots)\mathfrak{f}$ where b_i is either a symbol or a sequence of symbols.

We define the corresponding notion of binding a sequence of symbols $[s_1 \ldots s_n]$ to a value v in an environment ξ by

$$\xi\{[s_1, s_2, \ldots, s_n] \leftarrow v\} \underset{\mathrm{df}}{=} \xi\{s_1 \leftarrow 1^{\mathrm{st}}\, v, [s_2, \ldots, s_n] \leftarrow \mathrm{r}^{\mathrm{st}}\, v\}.$$

Thus we have

$$\langle \lambda[s_1, s_2, \ldots, s_n]\mathfrak{f} \mid \xi\rangle{}'\, v \gg \langle \mathfrak{f} \mid \xi\{[s_1, s_2,\ \ldots\ , s_n] \leftarrow v\rangle$$

IV.3.4. Dtree expressions

To provide a richer language for expressing facts about the evaluation relation and pfn application, we form expressions by repeated use of the dtree application operation and sequence operations. We call such expressions dtree expressions. Thus value variables and constants are dtree expressions and if $A, A_0, A_1 \ldots, A_n$ are dtree expressions then so are $A_0{}'A_1$, $[A_0, \ldots, A_n]$, $1^{\mathrm{st}}\,(A)$, $\mathrm{r}^{\mathrm{st}}\,(A)$, $A{\downarrow}_i$, etc. The evaluation relation is extended to dtree expressions by first evaluating subexpressions. Some example expressions and evaluations are

$$u \hookrightarrow v \leftrightarrow u = v$$

$$(\vartheta{}'u_0){}'u_1 \hookrightarrow v \leftrightarrow (\exists\vartheta_0)(\vartheta{}'u_0 \hookrightarrow \vartheta_0 \wedge \vartheta_0{}'u_1 \hookrightarrow v)$$

$$(\vartheta_0{}'u_0){}'(\vartheta_1{}'u_1) \hookrightarrow v \leftrightarrow (\exists\vartheta, u)(\vartheta_0{}'u_0 \hookrightarrow \vartheta \wedge \vartheta_1{}'u_1 \hookrightarrow u \wedge \vartheta{}'u \hookrightarrow v)$$

$$[(\vartheta{}'u_0){}'u_1, u] \hookrightarrow v \leftrightarrow (\exists v_0)((\vartheta{}'u_0){}'u_1 \hookrightarrow v_0 \wedge v = [v_0, u])$$

$$1^{\mathrm{st}}\,((\vartheta{}'u_0){}'u_1) \hookrightarrow v \leftrightarrow (\exists v_0)((\vartheta{}'u_0){}'u_1 \hookrightarrow v_0 \wedge v = 1^{\mathrm{st}}\, v_0)$$

The definitions of definedness and weak equality extend naturally to dtree expressions. For dtree expressions A and B,

$$\Downarrow A \underset{\mathrm{df}}{\equiv} (\exists v)(A \hookrightarrow v)$$

$$A \rightleftharpoons B \underset{\mathrm{df}}{\equiv} (\forall v)(A \hookrightarrow v \leftrightarrow B \hookrightarrow v)$$

To parallel the currying of multiple arguments in forms we write $\vartheta{}'(u_1,\ \ldots\ , u_n)$ for $(\ldots\ (\vartheta{}'u_1)\ \ldots){}'u_n$ and when $\vartheta{}'(u_1,\ \ldots\ , u_n)$ is defined, we write $\vartheta(u_1,\ \ldots\ , u_n)$ for the value.

IV.3.5. *n*-ary pfns and pfnls

It is useful to give names to classes of "higher order" pfns. The most frequently used notion is that of pfnl, named in analogy to functional. More generally we can talk about *n*-ary pfns - pfns that can meaningfully be applied to a list of *n* arguments.

▹ **Pfnl.** A pfn is called a *pfnl* if it maps any argument to a pfn. We use *Pfnl* to denote the set of pfns that are pfnls.

$$\varphi \in \mathit{Pfnl} \underset{\text{df}}{\equiv} (\forall v)(\exists \varphi_1)\varphi{}'\, v \hookrightarrow \varphi_1$$

▹ ***n*-ary Pfn.** Any pfn is a 1-ary pfn. For $n > 1$, φ is an *n*-ary pfn if $\varphi{}'\, v$ is defined with value an $n - 1$-ary pfn for all values v.

- Pfnls are just the 2-ary pfns
- $\langle \lambda(\mathsf{x}_1, \ldots, \mathsf{x}_n)\mathfrak{f} \mid \xi \rangle$ is an *n*-ary pfn for any form $\mathfrak{f}$ and environment ξ.

IV.3.6. Derived forms

In order to make definitions more compact and readable, we introduce some abbreviations for forms. ifmt expresses the test for the empty sequence positively. and, or, and not describe boolean operations. The pfn $\lambda(\mathsf{x})\mathsf{mt}$ is chosen as the canonical non-empty sequence, to represent "true". Any pfn would do.

$$\mathsf{ifmt}(\mathfrak{f}_{\mathsf{test}}, \mathfrak{f}_{\mathsf{mt}}, \mathfrak{f}_{\mathsf{nmt}}) \underset{\text{df}}{=} \mathsf{if}(\mathfrak{f}_{\mathsf{test}}, \mathfrak{f}_{\mathsf{nmt}}, \mathfrak{f}_{\mathsf{mt}})$$

$$\mathsf{or}(\mathfrak{f}_{\mathsf{lhs}}, \mathfrak{f}_{\mathsf{rhs}}) \underset{\text{df}}{=} \mathsf{if}(\mathfrak{f}_{\mathsf{lhs}}, \lambda(\mathsf{x})\mathsf{mt}, \mathsf{if}(\mathfrak{f}_{\mathsf{rhs}}, \lambda(\mathsf{x})\mathsf{mt}, \mathsf{mt}))$$

$$\mathsf{and}(\mathfrak{f}_{\mathsf{lhs}}, \mathfrak{f}_{\mathsf{rhs}}) \underset{\text{df}}{=} \mathsf{if}(\mathfrak{f}_{\mathsf{lhs}}, \mathsf{if}(\mathfrak{f}_{\mathsf{rhs}}, \lambda(\mathsf{x})\mathsf{mt}, \mathsf{mt}), \mathsf{mt})$$

$$\mathsf{not}(\mathfrak{f}) \underset{\text{df}}{=} \mathsf{if}(\mathfrak{f}, \mathsf{mt}, \lambda(\mathsf{x})\mathsf{mt})$$

Finally we introduce let which expresses the use of abstraction followed by application as a means of temporarily giving a name to the value of an expression. To evaluate $\mathsf{let}\{s_1 \leftarrow \mathfrak{f}_1, s_2 \leftarrow \mathfrak{f}_2\}\mathfrak{f}_{\mathsf{body}}$ in an environment ξ, the arguments $\mathfrak{f}_1$ and $\mathfrak{f}_2$ are each evaluated in ξ. If $\mathfrak{f}_{\mathrm{i}}$ returns value v_{i} for $i = 1, 2$ then $\mathfrak{f}_{\mathsf{body}}$ is evaluated in ξ extended by binding v_1 to s_1 and v_2 to s_2. In general we have

$$\mathsf{let}\{\ldots,\ b_i \leftarrow \mathfrak{f}_{\mathrm{i}}\ , \ldots\}\mathfrak{f}_{\mathsf{body}} \underset{\text{df}}{=} \{\lambda(\ldots,\ b_i\ , \ldots)\mathfrak{f}_{\mathsf{body}}\}(\ldots,\ \mathfrak{f}_{\mathrm{i}}\ , \ldots)$$

where b_i is a symbol or a sequence of symbols. As illustrated above, and continuing the convention introduced in Chapter I, we will often use $\{\ldots\}$ to set off the function part of an application form when the function part is a complex expression. This is merely to try to improve readability.

To illustrate the added notation we have the following simple and useful consequence of the definition of the closure formation operation.

■ **Forgetting lemma.** For x not free in $\mathfrak{f}_1$

$$\Downarrow\mathfrak{f}_0 \rightarrow \mathsf{let}\{\mathsf{x} \leftarrow \mathfrak{f}_0\}\mathfrak{f}_1 \rightleftharpoons \mathfrak{f}_1.$$

In terms of basic *Rum* concepts this fact is expressed by

$$(\forall\xi)(\Downarrow\langle\mathfrak{f}_0 \mid \xi\rangle \rightarrow \langle\mathsf{app}(\lambda(\mathsf{x})\mathfrak{f}_1, \mathfrak{f}_0) \mid \xi\rangle \rightleftharpoons \langle\mathfrak{f}_1 \mid \xi\rangle)$$

IV.4. A library of pfn definitions

IV.4.1. Algebraic combinators

Algebraic combinators are those whose functional behavior is given explicitly by algebraic equations. Following the notation of Barendregt [1981] we define

▷ $\mathsf{I} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x})\mathsf{x}$

▷ $\mathsf{K} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x},\mathsf{y})\mathsf{x}$

▷ $\mathsf{C1} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f},\mathsf{x})\mathsf{f}(\mathsf{x})$

▷ $\mathsf{B} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f},\mathsf{g},\mathsf{x})\mathsf{f}(\mathsf{g}(\mathsf{x}))$

▷ $\mathsf{S} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f},\mathsf{g},\mathsf{z})\mathsf{f}(\mathsf{z})(\mathsf{g}(\mathsf{z}))$

I is the identity pfn, K is the constant maker, B is the composition pfn, $\mathsf{C1}$ the Church numeral one (also the application pfn) and S is the substitution pfn. These pfns are characterized computationally by the following properties which are easily proved using the definitions and rules for computation.

■ **Computational behavior of algebraic combinator pfns**

(I.redt) $\mathsf{I}(\mathsf{x}) \gg \mathsf{x}$

(K.redt) $\mathsf{K}(\mathsf{x},\mathsf{y}) \gg \mathsf{x}$

(C1.redt) $\mathsf{C1}(\mathsf{f},\mathsf{x}) \gg \mathsf{f}(\mathsf{x})$

(B.redt) $\mathsf{B}(\mathsf{f},\mathsf{g},\mathsf{x}) \gg \mathsf{f}(\mathsf{g}(\mathsf{x}))$

(S.redt) $\mathsf{S}(\mathsf{f},\mathsf{g},\mathsf{z}) \gg \mathsf{f}(\mathsf{z},\mathsf{g}(\mathsf{z}))$

By (redt.ctree.eq) we can replace $\gg\!\!\succ$ by $\rightleftharpoons$ in the above theorem. Thus the algebraic combinator pfns satisfy the usual equations and more. Note that K, $\mathsf{C1}$, B and S are pfnls and for all u,v $\mathsf{B}(u,v)$ and $\mathsf{S}(u,v)$ are pfns. We use the infix notation $u \circ v$ for $\mathsf{B}(u,v)$.

■ **The S^m_n theorem.** For each $m > 0$, $n > 0$ let S^m_n be defined by

$$S^m_n \underset{\mathrm{df}}{=} \lambda(\mathsf{f})\lambda(\mathsf{x}_1,\ldots,\mathsf{x}_\mathsf{m})\lambda(\mathsf{y}_1,\ldots,\mathsf{y}_\mathsf{n})\mathsf{f}(\mathsf{x}_1,\ldots,\mathsf{x}_\mathsf{m},\mathsf{y}_1,\ldots,\mathsf{y}_\mathsf{n})$$

then for any $(n+m)$-ary pfn ϑ

$$\Downarrow S^m_n{}'(\vartheta, u_1 \ldots u_\mathrm{m}) \quad \text{and} \quad S^m_n(\vartheta, u_1 \ldots u_\mathrm{m})'(v_1 \ldots v_\mathrm{n}) \gg\!\!\succ \vartheta'(u_1 \ldots u_\mathrm{m}, v_1 \ldots v_\mathrm{n})$$

The S^m_n theorem for *Rum* is a trivial consequence of the definitions of n-ary pfn and pfn application. We have stated it to emphasize the connection between closure formation and parameter fixing.

IV.4.2. Recursion

The *Rum* recursion theorem says that for each pfnl ϑ there is a pfn that is the computationally minimal fixed point of ϑ and that these fixed points are computed uniformly by the recursion pfn Rec.

▷ $\mathsf{Rec1} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{g})\lambda(\mathsf{h})\lambda(\mathsf{x})\mathsf{g}(\mathsf{h}(\mathsf{h}),\mathsf{x})$

▷ $\mathsf{Rec} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f})\mathsf{Rec1}(\mathsf{f},\mathsf{Rec1}(\mathsf{f}))$

Rec is defined in terms of the auxiliary Rec1 simply to make the definition more readable. Rec1 is also a useful intermediary in writing out the proof of the recursion theorem.

■ **Recursion Theorem.**

(rec.pfnl) $\mathsf{Rec} \in \mathit{Pfnl}$

(rec.subc) $(\forall\vartheta \in \mathit{Pfnl})(\vartheta'(\mathsf{Rec}(\vartheta)) \prec \mathsf{Rec}(\vartheta)'v)$

(rec.redt) $(\forall\vartheta \in \mathit{Pfnl})(\mathsf{Rec}(\vartheta)'v \gg\!\!\succ \vartheta(\mathsf{Rec}(\vartheta))'v)$

Computational minimality is expressed by (rec.redt). (rec.subc) says that the application of a pfnl to its fixed point is a subcomputation of every application of the fixed point to an argument. The recursion theorem is an easy consequence of the computation rules and the definition of Rec.

■ **Recursion Corollary.** The usual fixed point property is an immediate corollary of (rec.redt). This can be expressed in terms of pfns and values (rec.fix.a) or in terms of forms (rec.fix.e).

(rec.fix.a) $\mathsf{Rec}(\vartheta)'v \rightleftharpoons \vartheta(\mathsf{Rec}(\vartheta))'v$

(rec.fix.e) $\mathsf{f} \rightleftharpoons \mathsf{Rec}(\lambda(\mathsf{f})\lambda(\mathsf{x}_1,\ldots \mathsf{x}_n)\mathfrak{f}_{\mathsf{body}}) \rightarrow \mathsf{f}(\mathsf{x}_1,\ldots \mathsf{x}_n) \rightleftharpoons \mathfrak{f}_{\mathsf{body}}$

■ **Non Recursion.** If f is not free in $\mathfrak{f}_{\mathsf{body}}$ then

(rec.noop) $\{\mathsf{Rec}(\lambda(\mathsf{f})\lambda(\mathsf{x})\mathfrak{f}_{\mathsf{body}})\}(\mathsf{x}) \rightleftharpoons \mathfrak{f}_{\mathsf{body}}$

This follows from the recursion corollary and the forgetting lemma.

Parameterized Recursion. Consider the equation

$$\mathsf{f}(\mathsf{g},\mathsf{x}) \rightleftharpoons \mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{h}(\mathsf{x}),\mathsf{f}(\mathsf{g},\mathsf{g}(\mathsf{x}))$$

Since g is a parameter – is not changed in the recursive call – this equation can be solved in two ways using Rec. One way is to pass g explicitly as an argument at each recursive call

(param.pass) $\mathsf{f} \rightleftharpoons \mathsf{Rec}(\lambda(\mathsf{f})\lambda(\mathsf{g},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{h}(\mathsf{x}),\mathsf{f}(\mathsf{g},\mathsf{g}(\mathsf{x}))))$

the other way is to treat g as a parameter of the recursion

(param.fix) $\mathsf{f} \rightleftharpoons \lambda(\mathsf{g})\mathsf{Rec}(\lambda(\mathsf{ff})\lambda(\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{h}(\mathsf{x}),\mathsf{ff}(\mathsf{g}(\mathsf{x}))))$

Extensionally the two definitions are the same.

$$\{\mathsf{Rec}(\lambda(\mathsf{f})\lambda(\mathsf{g},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{h}(\mathsf{x}),\mathsf{f}(\mathsf{g},\mathsf{g}(\mathsf{x}))))\}(\mathsf{g},\mathsf{x})$$
$$\rightleftharpoons \{\lambda(\mathsf{g})\mathsf{Rec}(\lambda(\mathsf{ff})\lambda(\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{h}(\mathsf{x}),\mathsf{ff}(\mathsf{g}(\mathsf{x}))))\}(\mathsf{g},\mathsf{x})$$

Computationally the two solutions differ in the amount of work done in argument passing (binding of symbols to extend environments). In the (param.pass) case g is rebound in each recursive call, while in (param.fix) case g is bound initially and carried in the environment.

Why not Church's Y? The algebraic combinatory pfns are direct analogs of the corresponding lambda calculus combinators, but Rec differs from the direct analogue to Church's Y-combinator, ChurchY (see Barendregt [1981] p.127).

▷ $\mathsf{Y1} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f})\lambda(\mathsf{h})\mathsf{f}(\mathsf{h}(\mathsf{h}))$

▷ $\mathsf{ChurchY} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f})\mathsf{Y1}(\mathsf{f})(\mathsf{Y1}(\mathsf{f}))$

This is necessary as the following theorem shows. The reason is the difference between *Rum* evaluation, which is call-by-value and lambda calculus reduction which is call-by-name.

■ ChurchY is everywhere undefined.

Proof: For any v, $\mathsf{ChurchY}'v \succ\!\!\succ \mathsf{Y1}(v)'\mathsf{Y1}(v)$ and $\mathsf{Y1}(v)'\mathsf{Y1}(v) \prec \mathsf{Y1}(v)'\mathsf{Y1}(v)$. Hence by (loop) we have $\neg\Downarrow(\mathsf{Y}'v)$.

IV.4.3. Recursion on sequences

Collection, tupling, bounded search and iteration along a sequence are examples of computation schemes based on recursion on finite sequences. These schemes are total in the sense that when instantiated using total functions for the function parameters, the function defined is total. Such schemes are represented uniformly in *Rum* by pfnls which when applied to pfns computing the function parameters give a pfn computing the corresponding instance of the scheme.

Collection. The collection scheme has a unary function parameter. The parameter is applied to each element of the argument sequence. The results are collected in a sequence and returned as the value. The pfnl Collect describes the collection scheme

▷ $\mathsf{Collect} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{co})\lambda[\mathsf{x},\mathsf{y}]\mathsf{ifmt}(\mathsf{x},\mathsf{mt},[\mathsf{f}(\mathsf{x}),\mathsf{co}(\mathsf{y})]))$

and satisfies

$$\mathsf{Collect}(\vartheta)'[a_1,\ldots,a_{\mathrm{n}}] \rightleftharpoons [\vartheta' a_1,\ldots,\vartheta' a_{\mathrm{n}}].$$

Tupling. Tupling is the "dual" to collecting. The tupling scheme has a sequence of unary functions as a parameter. Each element of the parameter sequence is applied to the given argument and the collected results are returned as the value. In *Rum* tupling is explicitly definable from collection.

▷ $\mathsf{Tuple} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{funs})\lambda(\mathsf{x})\mathsf{Collect}(\lambda(\mathsf{h})(\mathsf{h}(\mathsf{x})),\mathsf{funs})$

Tuple satisfies

$$\mathsf{Tuple}[\vartheta_1,\ldots,\vartheta_{\mathrm{n}}]'u \rightleftharpoons [\vartheta_1' u,\ldots,\vartheta_{\mathrm{n}}' u].$$

Bounded search. The bounded search scheme has two unary function parameters. The first function computes a predicate. It is applied to each element of an argument sequence until an element is found that satisfies the predicate. Then the second parameter is applied to that element and the result returned. If no such element exists then the empty sequence is returned. We use Some to describe bounded search.

▷ $\mathsf{Some} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{so})\lambda[\mathsf{x},\mathsf{y}]\mathsf{ifmt}(\mathsf{x},\mathsf{mt},\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{f}(\mathsf{x}),\mathsf{so}(\mathsf{y}))))$

Some satisfies

$$(\forall a \in u)\vartheta_{\mathrm{p}}' a \hookrightarrow \Box \rightarrow \mathsf{Some}(\vartheta_{\mathrm{p}},\vartheta_{\mathrm{f}})' u \hookrightarrow \Box$$

$$(\forall a \in u)\Downarrow\vartheta_{\mathrm{p}}' a \wedge (\exists a \in u)\vartheta_{\mathrm{p}}(a) \neq \Box \rightarrow (\exists a \in u)\mathsf{Some}(\vartheta_{\mathrm{p}},\vartheta_{\mathrm{f}})' u \succ\!\!\succ \vartheta_{\mathrm{f}}' a$$

In the definitions of Collect and Some we have used $\lambda[x,y]$ to name the first and remainder of the argument at the beginning of the computation, and used the fact that $1^{\mathrm{st}}\, v = \square \iff v = \square$.

Iteration along a sequence. This scheme has a binary function and an initial value as parameters. If applied to an empty argument sequence the initial value is returned. If applied to a non-empty argument sequence the function is applied to the first element of the argument and the result of iterating along the rest of the argument sequence. SeqIt describes iteration along a sequence.

$$\triangleright \quad \mathsf{SeqIt} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{f},\mathsf{z})\mathsf{Rec}(\lambda(\mathsf{si})\lambda(\mathsf{x})\mathsf{ifmt}(\mathsf{x},\mathsf{z},\mathsf{f}(\mathsf{fst}(\mathsf{x}),\mathsf{si}(\mathsf{rst}(\mathsf{x})))))$$

SeqIt satisfies

$$\mathsf{SeqIt}'(\vartheta, u, [a_1, \dots a_{\mathrm{m}}]) \rightleftharpoons \vartheta'(a_1, \dots, \vartheta'(a_{\mathrm{m}}, u)\dots)$$

Exercise 1. Tuple was defined using Collect to illustrate the expressive power of collection in the presence of pfns. Derive an equation for Tuple that does not use Collect and prove that the solution of the derived equation using Rec computes the tupling pfn. (This is even simpler than the TprodC derivation §II.3 as there is no need to introduce added parameters.)

IV.4.4. Remarks

- The computational facts for the algebraic combinator pfns, the computational minimality of the recursion pfn, and computation induction illustrate some of the features of *Rum* in contrast to those of computation theories based on codes as elements of the computation domain or recursion theories based on lambda reduction (see §I.5). In a nutshell, we have the strong form of the closure conditions required for a computation theory and at the same time the corresponding functionals are defined extensionally using abstraction.

- (rec.redt) is stronger than (rec.fix). Combined with computation induction, (rec.redt) is an important tool for proving properties of recursively defined pfns. Recursion induction (McCarthy [1963]), subgoal induction (Morris and Wegbreit [1976]), McCarthy's minimization schema (a scheme formalizing Kleene's first recursion theorem – see Cartwright and McCarthy [1979]), and many other instances of "Scott induction" follow from (rec.redt) and (CI). "Scott induction" is an induction principle derived from the least fixed point theorem for extensional models of the lambda-calculus (Scott [1976]).

- The example of alternative descriptions of parameterized recursion relies on the fact that the pfnl has the form of an ordinary well typed functional. Intuitively it would seem that moving parameters across the recursion boundary should in general preserve the function computed. We will see in §VI.2 that this is indeed the case.

- SeqIt is so named in analogy to a functional *lit* defining iteration along a list and studied by Gordon [1973].

- Collection, tupling and iteration along sequences correspond to the constructs *apply-to-all*, *construct*, and *insert* in functional programming (Backus [1978]). Collection also corresponds to the finite collection schema use to formalize fragments of arithmetic and is a finitary analog to the collection principle of set theory.

IV.5. Computing with S-expressions

Now we fix the data structure to be the S-expression structure described in §III.5 and define some additional data operations and constants. Many of these are basic definitions that will be used in later examples. In addition they serve to illustrate further the use of sequences and of the library of pfns built in §4.

The global S-expression context associates each S-expression operation to the corresponding S-expression operation symbol. The same identifier will be used to denote both an S-expression operation and its corresponding symbol. Thus PairMk denotes both the pairing operation and the symbol whose value is the pairing operation.

IV.5.1. Standard S-expression operations and constants

The Lisp name for the empty list in Nil. The projection operations for pairs, Car and Cdr, are defined from PairUn using fst and rst and Atom is the opposite of PairP (on the S-expression domain $\mathbb{D}_{\mathrm{sexp}}$).

▷ $\mathsf{0} \overset{\mathrm{df}}{\leftrightarrow} \mathsf{ZeroMk}[]$

▷ $\mathsf{Nil} \overset{\mathrm{df}}{\leftrightarrow} \mathsf{MtlMk}[]$

▷ $\mathsf{Car} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{x})\mathsf{fst}(\mathsf{PairUn}(\mathsf{x}))$

▷ $\mathsf{Cdr} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{x})\mathsf{rst}(\mathsf{PairUn}(\mathsf{x}))$

▷ $\mathsf{Atom} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{x})\mathsf{if}(\mathsf{PairP}(\mathsf{x}),\mathsf{mt},\mathsf{x})$

For S-expressions a,a_1,a_2 we have

$$\mathsf{Car}(\mathsf{PairMk}[a_1,a_2]) = a_1, \qquad \mathsf{Cdr}(\mathsf{PairMk}[a_1,a_2]) = a_2$$

$$\mathsf{Atom}(\mathsf{PairMk}[a_1,a_2]) = \square, \qquad \mathsf{PairUn}'(a) = \mathsf{Tuple}([\mathsf{Car},\mathsf{Cdr}],a)$$

Exercise 1. Since $\mathbb{D}_{\mathrm{sexp}}$ 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. SexpEq satisfies

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

SexpEq is not specified for non-S-expression arguments.

Exercise 2. The computable functions on integers are definable as *Rum* pfns in the usual way using Rec to solve recursion equations. Define addition (+), multiplication (*), less-than (Lessp), and the function that returns the maximum of two integers (Max).

IV.5.2. Using multi-ary operations

StrConc is string concatenation. It is defined using sequence concatenation and the operations StrUn and StrMk for interconversion of strings and sequences.

▷ $\mathsf{StrConc} \overset{\mathrm{df}}{\longleftrightarrow} \lambda[\mathsf{x},\mathsf{y}]\mathsf{StrMk}[\mathsf{StrUn}(\mathsf{x}),\mathsf{StrUn}(\mathsf{y})]$

For integers z_i we have

$$\mathsf{StrConc}[\text{"}z_1,\ldots,z_m\text{"},\text{"}z_{m+1},\ldots,z_{m+n}\text{"}] = \text{"}z_1,\ldots,z_{m+n}\text{"}$$

Using SeqIt we define a pfn ListMk, analogous to the StrMk data operation, that maps sequences of S-expressions to lists. The inverse of ListMk is ListUn, defined by a simple recursive definition.

▷ $\mathsf{ListMk} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x})\mathsf{SeqIt}(\mathsf{PairMk},\mathsf{Nil},\mathsf{x})$

▷ $\mathsf{ListUn} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{ListUn})\lambda(\mathsf{x})\mathsf{if}(\mathsf{MtIP}(\mathsf{x}),\mathsf{Nil},[\mathsf{Car}(\mathsf{x}),\mathsf{ListUn}(\mathsf{Cdr}(\mathsf{x}))]))$

This definition of the list making operation is a direct implementation of the informal notation we gave for lists in our presentation of the S-expression data structure (§III.5) and we have for S-expressions a_i

$$\mathsf{ListMk}'[a_1,\ldots,a_n] \hookrightarrow {<}a_1,\ldots a_n{>}$$

$$\mathsf{ListMk}'(\mathsf{ListUn}({<}a_1,\ldots a_n{>})) \rightleftharpoons {<}a_1,\ldots a_n{>}$$

$$\mathsf{ListUn}'(\mathsf{ListMk}([a_1,\ldots a_n])) \rightleftharpoons [a_1,\ldots a_n]$$

For lists l_0, l_1 we will use $l_0 \diamond l_1$ to denote the concatenation. Thus

$$l_0 \diamond l_1 = \mathsf{ListMK}[\mathsf{ListUn}(l_0), \mathsf{ListUn}(l_1)]$$

Two additional interesting operations that combine lists and multi-ary operations are ListExtend and its inverse ListAfter. ListExtend takes a sequence and a list and extends the list, adding the elements of the sequence to the list. ListAfter takes a sequence and a list and, if the sequence corresponds to an initial segment of the list the remaining tail of the list is returned. Otherwise the empty sequence is returned.

▷ $\mathsf{ListExtend} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x},\mathsf{y})\mathsf{SeqIt}(\mathsf{PairMk},\mathsf{y},\mathsf{x})$

▷ $\mathsf{ListAfter} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{ListAfter})\lambda(\mathsf{x},\mathsf{y})$

$\mathsf{ifmt}(\mathsf{x},\mathsf{y},$

$\mathsf{if}(\mathsf{MtlP}(\mathsf{y}),\mathsf{mt},$

$\mathsf{if}(\mathsf{SexpEq}(\mathsf{fst}(\mathsf{x}),\mathsf{Car}(\mathsf{y})),\mathsf{ListAfter}(\mathsf{rst}(\mathsf{x}),\mathsf{Cdr}(\mathsf{y})),\mathsf{mt}))))$

Note that ListMk is just ListExtend with the list parameter fixed to be Nil. For S-expressions a_i and lists l we have

$$\mathsf{ListExtend}'([a_1,\dots a_n],l) \rightleftharpoons \mathsf{ListMk}'([a_1,\dots a_n,\mathsf{ListUn}(l)]))$$

$$\mathsf{ListAfter}'([a_1,\dots a_n],\mathsf{ListExtend}([a_1,\dots a_n],l)) \rightleftharpoons l$$

IV.5.3. Tprod revisited

We have now defined most of the concepts needed to give precise interpretation to the definitions and statements regarding tree-structured computation in the tree product examples of Chapter II. What remains is to explain the recursive definitions of the pfns as used there. Definitions of the form $\mathsf{f}(\mathsf{x}_1 \dots \mathsf{x}_n) \leftarrow \mathfrak{f}_{\mathsf{body}}$ define f to be the pfn which is the minimal fixed point of the corresponding pfnl $\lambda(\mathsf{f})\lambda(\mathsf{x}_1 .. \mathsf{x}_n)\mathfrak{f}_{\mathsf{body}}$. For example,

▷ $\mathsf{Tprod}(\mathsf{x}) \leftarrow \mathsf{if}(\mathsf{Atom}(\mathsf{x}),\mathsf{x},\mathsf{Tprod}(\mathsf{Car}(\mathsf{x})) * \mathsf{Tprod}(\mathsf{Cdr}(\mathsf{x})))$

is an abbreviation for

▷ $\mathsf{Tprod} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{Tprod})\lambda(\mathsf{x})\mathsf{if}(\mathsf{Atom}(\mathsf{x}),\mathsf{x},*[\mathsf{Tprod}(\mathsf{Car}(\mathsf{x})),\mathsf{Tprod}(\mathsf{Cdr}(\mathsf{x}))]))$

In the examples we used infix notation for operations such as for $+$ and $*$. For the official work in $\mathcal{R}um$ we have not bothered to introduce infix notations.

At this point the reader should be able to fill in all the details for definitions and extensional statements about the pfns describing tree-structured computation. Using the informal description of computation trees the intensional statements can also be understood intuitively. The definitions of Appendix A and the further examples in Appendix B show how these statements can be made precise.

IV.6. Streams and stream operations

Streams were described in §I.2 as 0-ary function-like objects with internal state that present possibly infinite sequences. In *Rum* there is no notion of internal state, so we must explicitly keep track of the information about the rest of a stream. Thus a stream is a pfn ϑ that when queried (applied to the empty sequence) either returns the empty sequence or an element and a stream. We call this the *stream condition*. In the first case, ϑ is the empty stream, and the empty sequence is the *end-of-stream* signal. In the second case, ϑ is a non-empty stream, the element returned is the first element of the stream, and the stream returned is the rest of the stream. Two examples of streams are: the pfn MtStream which is an empty stream and the pfn Streamify that turns a sequence into a stream of its elements.

$$\triangleright \qquad \mathsf{MtStream} \overset{\mathrm{df}}{\longleftrightarrow} \lambda()\mathsf{mt}$$

$$\triangleright \qquad \mathsf{Streamify} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{s})\lambda[\mathsf{x},\mathsf{y}]\lambda()\mathsf{ifmt}(\mathsf{x},\mathsf{mt},[\mathsf{x},\mathsf{s}(\mathsf{y})]))$$

We used $\lambda()\mathfrak{f}$ in the above definitions as an abbreviation for $\lambda(s)\mathfrak{f}$ where s is some symbol not appearing free in $\mathfrak{f}$, for example z in the above cases.

We let *Stream* denote the set of pfns that satisfy the stream condition given above. More precisely a *stream set* is any set of pfns Φ that satisfies (Sc).

$$\text{(Sc)} \qquad \vartheta \in \Phi \;\leftrightarrow\; \vartheta{}'\square \hookrightarrow \square \vee (\exists a, \vartheta_1 \in \Phi)\vartheta{}'\square \hookrightarrow [a, \vartheta_1]$$

Since the right hand side of (Sc) is monotone in Φ there is a least stream set $Stream^{\omega}$ – the set inductively generated by the corresponding monotone operation. $Stream^{\omega}$ consists of the finite streams and contains all empty streams and Streamify(u) for any u. The union of any non-empty set of stream sets is a stream set. Hence there is a maximum stream set. This is what we called *Stream*. *Stream* can also be given by a shrinking induction [Moschovakis 1975], i.e. as the compliment of an inductively defined set of pfns.

IV.6.1. Stream elements, tails, length, and equality

There are two sequences associated with each stream – the sequence of stream elements and the sequence of stream remainders. These are given by the n-th element and n-th tail operations on streams. We call the pair [n-th element, n-th tail] the n-th iterate. The length of a stream is the number of elements – possibly infinite. Two streams are equal (as streams) if they have the same n-th elements for all n. These notions are made precise by the following definitions.

n-th iterate, element, and tail. The n-th iterate operation is computed by the pfn NthItr and the n-th element and n-th tail operations are computed by the pfns NthElt and $\mathsf{NthTail}$.

$$\triangleright \quad \mathsf{NthItr} \overset{\mathrm{df}}{\hookleftarrow} \mathsf{Rec}(\lambda(\mathsf{NthItr})\lambda(\mathsf{s},\mathsf{n}) \\ \mathsf{let}\{[\mathsf{x},\mathsf{t}] \leftarrow \mathsf{s}[]\} \\ \mathsf{if}(\mathsf{Zerop}(\mathsf{n}), [\mathsf{x},\mathsf{t}], \mathsf{NthItr}(\mathsf{t}, \mathsf{Sub1}(\mathsf{n}))))$$

$$\triangleright \quad \mathsf{NthElt} \overset{\mathrm{df}}{\hookleftarrow} \lambda(\mathsf{s},\mathsf{n})\mathsf{let}\{[\mathsf{x},\mathsf{t}] \leftarrow \mathsf{NthItr}(\mathsf{s},\mathsf{n})\}\mathsf{x}$$

$$\triangleright \quad \mathsf{NthTail} \overset{\mathrm{df}}{\hookleftarrow} \lambda(\mathsf{s},\mathsf{n})\mathsf{let}\{[\mathsf{x},\mathsf{t}] \leftarrow \mathsf{NthItr}(\mathsf{s},\mathsf{n})\}\mathsf{t}$$

For a stream ϑ and a number n we write

$$\vartheta^{(n)} \underset{\mathrm{df}}{=} \mathsf{NthElt}(\vartheta, n) \quad \text{and} \quad \vartheta^{>n} \underset{\mathrm{df}}{=} \mathsf{NthTail}(\vartheta, n)$$

Note that for any stream ϑ and any number n NthItr is defined and we have

$$[\vartheta^{(0)}, \vartheta^{>0}] = \vartheta(\square) \quad \text{and} \quad [\vartheta^{(n+1)}, \vartheta^{>n+1}] = \begin{cases} \vartheta^{>n}(\square) & \text{if } \vartheta^{(n)} \neq \square \\ \square & \text{if } \vartheta^{(n)} = \square \end{cases}$$

$\triangleright$ **Stream length.** The length of a stream $|\vartheta|_s$ is the least number n such that $\vartheta^{(n)} = \square$ if such a number exists. Otherwise the length is defined to be ∞.

$$|\vartheta|_s = \begin{cases} \mu(n)\vartheta^{(n)} = \square & \text{if } (\exists n)\vartheta^{(n)} = \square \\ \infty & \text{otherwise} \end{cases}$$

$\triangleright$ **Stream equality.** Two streams ϑ_0, ϑ_1 are *stream-equal* (written $\vartheta_0 \overset{s}{=} \vartheta_1$) if they have the same n-th element for all n.

$$\vartheta_0 \overset{s}{=} \vartheta_1 \underset{\mathrm{df}}{\equiv} (\forall n)\vartheta_0^{(n)} = \vartheta_1^{(n)}$$

■ **About stream elements, tails, length and equality.** For streams ϑ, ϑ_0, ϑ_1

$$(\forall n < |\,u\,|)(\mathsf{Streamify}(u)^{(n)} = u\!\downarrow_n)$$

$$n < |\vartheta|_s \rightarrow \vartheta^{(n)} \in \mathbf{V} \wedge \vartheta^{>n} \in \mathit{Stream}$$

$$n \geq |\vartheta|_s \rightarrow \vartheta^{(n)} = \square \wedge \vartheta^{>n} = \square$$

$$n + k < |\vartheta|_s \rightarrow \vartheta^{(n+k+1)} = (\vartheta^{>n})^{(k)}$$

$$|\mathsf{MtStream}|_s = 0 \quad \text{and} \quad |\mathsf{Streamify}(v)|_s = |v|$$

$$\vartheta_0 \overset{s}{=} \vartheta_1 \rightarrow |\vartheta_0|_s = |\vartheta_1|_s$$

$$\vartheta_0 \overset{s}{=} \vartheta_1 \wedge n < |\vartheta_0|_s \rightarrow (\vartheta_0)^{>n} \overset{s}{=} (\vartheta_1)^{>n}$$

These properties are easy to check from the definitions.

IV.6.2. Stream combination operations

New streams can be made from given streams by stream combination operations. Some examples are concatenation, filtering, merging, and mapping.

Concatenation

The concatenation of two streams is a stream that generates elements from the first stream until the end is reached, then generates elements from the second stream. StreamConc describes the concatenation operation.

$$\triangleright \quad \mathsf{StreamConc} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{StreamConc})\lambda(\mathsf{s0},\mathsf{s1}) \\ \lambda()\mathsf{let}\{[\mathsf{x},\mathsf{s0}] \leftarrow \mathsf{s0}[]\}\mathsf{if}(\mathsf{x},[\mathsf{x},\mathsf{StreamConc}(\mathsf{s0},\mathsf{s1})],\mathsf{s1}[])$$

We will write $\vartheta_0 \diamond \vartheta_1$ for $\mathsf{StreamConc}(\vartheta_0, \vartheta_1)$. Clearly if ϑ_0 and ϑ_1 are streams then $\vartheta_0 \diamond \vartheta_1$ is a stream.

■ **Properties of stream concatenation.** The following facts show that stream concatenation has the same properties as concatenation of other sequence like objects. For streams ϑ_0, ϑ_1, ϑ_2 we have

$$|\vartheta_0|_s = n_0 \wedge |\vartheta_1|_s = n_1 \to |\vartheta_0 \diamond \vartheta_1|_s = n_0 + n_1$$

$$|\vartheta_0|_s = \infty \vee |\vartheta_1|_s = \infty \to |\vartheta_0 \diamond \vartheta_1|_s = \infty$$

$$n < |\vartheta_0|_s \to (\vartheta_0 \diamond \vartheta_1)^{(n)} = \vartheta_0^{(n)} \wedge (\vartheta_0 \diamond \vartheta_1)^{>n} = \vartheta_0^{>n} \diamond \vartheta_1$$

$$|\vartheta_0|_s = m \wedge m \leq n \to (\vartheta_0 \diamond \vartheta_1)^{(n)} = \vartheta_1^{(n-m)} \wedge (\vartheta_0 \diamond \vartheta_1)^{>n} = \vartheta_0^{>(n-m)}$$

$$\vartheta_0 \overset{s}{=} \mathsf{MtStream} \diamond \vartheta_0 \overset{s}{=} \vartheta_0 \diamond \mathsf{MtStream}$$

$$(\vartheta_0 \diamond \vartheta_1) \diamond \vartheta_2 \overset{s}{=} \vartheta_0 \diamond (\vartheta_1 \diamond \vartheta_2)$$

$$\mathsf{Streamify}(u) \diamond \mathsf{Streamify}(v) \overset{s}{=} \mathsf{Streamify}([u, v])$$

Filtering

A *filter* takes a test and a stream and produces a stream in which the elements failing the test are filtered out. The pfn Filter describes the filter operation.

$$\triangleright \quad \mathsf{Filter} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{p})\mathsf{Rec}(\lambda(\mathsf{fil})\lambda(\mathsf{s})\lambda() \\ \mathsf{let}\{[\mathsf{x},\mathsf{s}] \leftarrow \mathsf{s}[]\}\mathsf{ifmt}(\mathsf{x},\mathsf{mt},\mathsf{if}(\mathsf{p}(\mathsf{x}),[\mathsf{x},\mathsf{fil}(\mathsf{s})],\mathsf{fil}(\mathsf{s})[])))$$

The key computational facts for a stream ϑ filtered by a test φ are

$$\mathsf{Filter}(\varphi,\vartheta)\,'\square \begin{cases} \gg\!\!\!\to \mathsf{Filter}(\varphi,\vartheta_1)\,'\square & \text{if } \vartheta\,'\square \hookrightarrow [a,\vartheta_1] \quad \text{and} \quad \varphi\,'a \hookrightarrow \square \\ \hookrightarrow [a,\mathsf{Filter}(\varphi,\vartheta_1)] & \text{if } \vartheta\,'\square \hookrightarrow [a,\vartheta_1] \quad \text{and} \quad \varphi\,'a \hookrightarrow v \neq \square \\ \hookrightarrow \square & \text{if } \vartheta\,'\square \hookrightarrow \square \end{cases}$$

■ **Filter distributes over concatenation.** For streams ϑ_0, ϑ_1

$$\mathsf{Filter}(\varphi,\vartheta_0 \diamond \vartheta_1) \overset{s}{=} \mathsf{Filter}(\varphi,\vartheta_0) \diamond \mathsf{Filter}(\varphi,\vartheta_1)$$

We can further characterize the filter operation by defining the index of the n-th element of a stream ϑ satisfying a predicate Φ. $Ix(\Phi,\vartheta,n)$, is defined by induction on numbers

$$Ix(\Phi,\vartheta,0) = \mu(n)\Phi(\vartheta^{(n)})$$
$$Ix(\Phi,\vartheta,m+1) = \mu(n)(n > Ix(\Phi,\vartheta)^{(m)} \wedge \Phi(\vartheta^{(n)}))$$

If Φ is computed by φ_Φ then

$$\mathsf{Filter}(\varphi_\Phi,\vartheta)^{(n)} = \vartheta^{(Ix(\Phi,\vartheta,n))}$$

Exercise 1. Define a pfn describing a merge operation on streams. The $2n$-th element of the the merge is the n-th element of the first stream, the $(2n+1)$-th element of the merge is the n-th element of the second stream.

Exercise 2. Define pfns StreamMap1, StreamMap2, and StreamMapS describing mapping operations on streams. For pfns φ defined on $\mathbf{V}$ and streams ϑ, the n-th element of StreamMap1(φ,ϑ) is the result of applying φ to the n-th element of ϑ. For pfns φ defined on $[\mathbf{V}\times\mathbf{V}]$ and streams ϑ_0, ϑ_1, the n-th element of StreamMap2$(\varphi,\vartheta_0,\vartheta_1)$ is the result of applying φ to the n-th element of ϑ_0 and the n-th element of ϑ_1. For pfns φ defined on $\mathbf{V}^*$ and any sequence of streams $[\vartheta_0\ldots\vartheta k]$, the n-th element of StreamMapS$(\varphi,[\vartheta_0\ldots\vartheta k])$ is the result of applying φ to the sequence of n-th elements of $[\vartheta_0\ldots\vartheta k]$.

Exercise 3. Show that unary mapping distributes over concatenation. What about binary and multi-ary mapping?

IV.6.3. Remarks

• **Other work** A representation of streams similar to the *Rum* representation was first used by Landin [1965] where streams were introduced to represent control lists of Algol. A control list is a list of expressions that describes a list of items to iterate through. Items of a control list should be computed as needed, not all at once. Landin treated only finite streams and gave operators analogous to normal list operations for construction and selection. Burge [1970] extends this idea to possibly infinite streams. He presents sequences as functions that return a pair consisting of the next element of the sequence and a function giving the rest of the sequence. Examples of programs computing a variety of operations involving such sequences are given. Burge does not give a formal definition of the set of streams (sequences in his terminology) nor does he state any mathematical properties of streams or the operations on streams. Both Landin and Burge remark on analogy of streams to Conway's notion of co-routine, but the connection is not elaborated.

• **Elaborations and generalizations** There are many ways to extend the basic ideas underlying the notion of stream in *Rum*. For example, one could allow partial streams rather than requiring that application of a stream to the empty sequence be defined. One could also allow arbitrary sequences to be produced as stream elements. A more substantial generalization arises when we relax the restriction that the behavior of a stream be completely determined by repeated application to the empty sequence. We say a pfn ϑ is a parameterized stream if for any element a of the computation domain ϑ applied to a returns either the empty sequence (signifying the end) or the next stream element and a parameterized stream (the remainder of the stream). A parameterized stream can be thought of as presenting a function from sequences to sequences. To see this we generalize the n-th element and n-th tail operations. For a parameterized stream ϑ and a function h mapping numbers to elements of the computation domain (the input sequence) define $\vartheta^{(h,n)}$, the n-th element of ϑ for input sequence φ, and $\vartheta^{(h,>n)}$, the n-th tail of ϑ for input sequence h, by induction on numbers as follows.

$$[\vartheta^{(h,0)}, \vartheta^{(h,>0)}] = \vartheta(h(0))$$

$$[\vartheta^{(h,n+1)}, \vartheta^{(h,>n+1)}] = \begin{cases} \vartheta^{(h,>n)}(h(n+1)) & \text{if } \vartheta^{(h,n)} \neq \square \\ \square & \text{if } \vartheta^{(h,n)} = \square \end{cases}$$

Then $\lambda(h,n)\vartheta^{(h,n)}$ is the function from sequences to sequences presented by ϑ. This gives a small taste of the many possibilities for generalizing streams.

Chapter V. Sequential computation in Rum

Now we turn our attention to sequential computation in *Rum*. Such computations are carried out by a process of generating sequences of computation states. A computation state consists of a continuation and a current task. The computation primitives are those for tree-structured computation together with a new primitive for noting (binding) the current continuation. Application is the mechanism for resuming a noted continuation. The structure of computation states and the rules for generating computation sequences are derived naturally from the structure of computation stages and the rules for generating computation trees. The tree structure of t-*Rum* computations is preserved in the computation sequences and many of the notions used in expressing properties of tree-structured computation can be interpreted as properties of sequential computation. In §1 the additional objects used to describe and carry out sequential computations are defined. In §2 the rules for generating computation sequences are presented. Basic theorems about sequential computation and the relation of sequential computation to tree-structured computation are given in §3. The remaining two sections give examples of programming and proving using continuations. In §4 continuations are used to construct an *until loop* from a *do-forever loop*. This is proved equivalent to the usual recursive definition of the until loop. In §5 we show how co-routine mechanisms can be represented in *Rum*, give some basic properties of co-routine resumption, and prove a simple co-routine correct.

The world of sequential computation is denoted by s-*Rum*. In this chapter we will be mainly working in the world of sequential computation and variables and domain symbols are to be interpreted in this world. Thus $\mathbb{F}$ is interpreted as s-$\mathbb{F}$, $\mathfrak{f}$ ranges over forms in s-$\mathbb{F}$, etc. The qualifier s- will be omitted except when comparing the features of t-*Rum* with those of s-*Rum*.

V.1. Objects for sequential computation

The objects of the s-*Rum* world include those of the t-*Rum* world. In addition, there are two new semantic domains – continuations and states. There is a new sort of form for noting continuations, a new sort of dtree for continuation application, and continuations are added to the computation domain. The objects of s-*Rum* are generated by adding appropriate clauses to the rules for generating the objects of t-*Rum*.

V.1.1. Forms describing sequential computation

As for tree-structured computation, the forms describing sequential computation are generated freely from the given set of symbols by constructions corresponding to the computation primitives. The construction rules are those given in Figure 10 (§IV.1) together with the note- construction

(note) $\mathsf{note}(s_{\text{cont}})\mathfrak{f}_{\text{body}}$ note the calling context

note is a binding construct analogous to the λ construct. Free occurrences of s in $\mathfrak{f}$ are bound by the outer note-construct in $\mathsf{note}(s)\mathfrak{f}$.

V.1.2. Semantic domains for sequential computation

The semantic domains for sequential computation are dtrees, pfns, environments, the computation domain, continuations, and states. These domains are generated from the given data and data operations and forms for sequential computation by a mutual inductive definition and are (isomorphic to) the minimal solutions to the following equations modulo the additional rules for equality ($\overset{\delta}{=}$).

(dtree) $\mathbb{D}t \sim (\mathbb{F} \times \mathbb{E})_{\overset{\delta}{=}} \oplus (\mathbb{O} \times \mathbb{D}^*) \oplus (\mathbb{C}o \times \mathbf{V}^*)$

(pfn) $\mathbb{P} \sim (\mathbb{S}y \times \mathbb{F} \times \mathbb{E})_{\overset{\delta}{=}}$

(environment) $\mathbf{E} \sim [\mathbb{S}y \ast\!\succ \mathbf{V}^*]$

(computation domain) $\mathbf{V} \sim \mathbb{D} \oplus \mathbb{O} \oplus \mathbb{P} \oplus \mathbb{C}o$

(continuation) $\mathbb{C}o \sim \{\mathsf{Id}\} \oplus \ldots$ % see Figure 15

(computation state) $\mathbb{S}t \sim (\mathbb{C}o \times \mathbb{D}t) \oplus (\mathbb{C}o \times \mathbf{V}^*)$

The equations for dtrees, pfns, environments, and values are discussed below. States and continuations are discussed in the next subsection.

- **Dtrees.** The three summands in the equation for dtrees correspond to closure formation, data operation application, and continuation application. Closure formation and data operation application were discussed in §IV.1. For each continuation γ and each value v, $\gamma' v$ is a continuation application dtree. $\gamma' v$ describes resuming computation in the context represented by γ with v the value returned. As before two dtrees are equal only if they are of the same sort. Two continuation application dtrees are equal iff the corresponding components are equal.

- **Pfns, environments and values.** Pfns and environments for sequential computation are generated in the same manner as in t-$\mathcal{R}um$. The computation domain for s-$\mathcal{R}um$ contains continuations as well as data, data operations and pfns.

We say that an expression $v_{\mathfrak{f}}\,'\,v_{\mathrm{a}}$ is *well-formed* if $v_{\mathfrak{f}}$ is a data operation and v_{a} is a data sequence or if $v_{\mathfrak{f}}$ is pfn or continuation. This extends the definition of well-formed given in §IV.1 to allow continuation application. Recall that $\langle\lambda(s)\mathfrak{f} \mid \xi\rangle\,'\,v$ is defined as $\langle\mathfrak{f} \mid \xi\{s \leftarrow v\}\rangle$.

V.1.3. Transforming computation stages into computation states

The structure of computation states and the rules for sequential computation are obtained by transforming the process of generating computation trees into one of generating computation sequences. The idea is to chose a strategy for generating computation trees which at each stage uniquely determines the node at which an extension can be made and the rule to apply (if any). The strategy we have chosen for generating computation trees corresponds to completing leftmost subcomputations first. This strategy can be built into the rules for generating computation trees by modifying the rules so that at any stage there is at most one node where a rule can be applied, and at most one rule that applies. All that is needed is to force the fun-subcomputation of an app-dtree to be completed before the arg-subcomputation is begun and to force the lhs-subcomputation of a cart-dtree to be completed before the rhs-subcomputation is begun. For example, the rule (app.arg) becomes

$$(\text{app.arg})^{*} \qquad \langle\mathfrak{f}_{\text{fun}} \mid \xi\rangle \hookrightarrow v_{\text{fun}} \;\rightarrow\; \langle\mathfrak{f}_{\text{arg}} \mid \xi\rangle \prec_{\iota} \langle\mathsf{app}(\mathfrak{f}_{\text{fun}},\mathfrak{f}_{\text{arg}}) \mid \xi\rangle$$

• **States.** Computation stages are represented by computation states. Using the modified rules, at each stage in a computation there is a *current node* and a *current context* – the tree surrounding the current node. The current context is represented by a continuation. If the rule that applies at the current node is a begin rule then the state representing this stage is the begin state $\gamma \triangledown \mathfrak{d}$ (read begin computation of $\mathfrak{d}$ with continuation γ) where γ is the continuation representing the current context and $\mathfrak{d}$ is the dtree label at the current node. If the rule that applies at the current node is a return rule then the state representing this stage is the return state $\gamma \vartriangle v$ (read return v to qco!) where γ is the continuation representing the current context and v the value to be returned.

States are generated freely from continuations, dtrees and values by begin and return constructions corresponding to the two summands in the equation for states.

$$\mathbb{S}t = (\mathbb{C}o \times \mathbb{D}t) \oplus (\mathbb{C}o \times \mathbf{V}^{*})$$

The sign $\triangledown$ in $\gamma \triangledown \mathfrak{d}$ is intended to suggest going down in the computation tree while the sign $\vartriangle$ in $\gamma \vartriangle v$ is intended to suggest going up in the computation tree.

• **Continuations.** Continuations are generated from forms, environments, and values according to the constructions shown in Figure 15. The continuation representing the current context of a computation stage generated by the modified rules represents that portion of the context that must be remembered in order to complete the computation when a value is returned. The identity continuation Id represents the empty context – the context of an initial or final stage. $\gamma \circ \langle \mathsf{Ifi}(\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}) \mid \xi\rangle)$ is the continuation representing the context of the test-subcomputation $\langle \mathfrak{f}_{\mathrm{test}} \mid \xi\rangle$ of $\langle \mathsf{if}(\mathfrak{f}_{\mathrm{test}},\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}) \mid \xi\rangle$ computed in a context represented by γ. $\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle$ is the continuation for the fun-subcomputation $\langle \mathfrak{f}_{\mathrm{fun}} \mid \xi\rangle$ of $\langle \mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle$ with continuation γ and, if the fun-subcomputation returns v_{fun}, $\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}})$ is the continuation for the arg-subcomputation $\langle \mathfrak{f}_{\mathrm{arg}} \mid \xi\rangle$. The continuation for the lhs-subcomputation $\langle \mathfrak{f}_{\mathrm{lhs}} \mid \xi\rangle$ of $\langle \mathsf{cart}(\mathfrak{f}_{\mathrm{lhs}},\mathfrak{f}_{\mathrm{rhs}}) \mid \xi\rangle$ with continuation γ is $\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi\rangle$ and, if the lhs-subcomputation returns v_{lhs}, the continuation for the rhs-subcomputation $\langle \mathfrak{f}_{\mathrm{rhs}} \mid \xi\rangle$ is $\gamma \circ \mathsf{Cartc}(v_{\mathrm{lhs}})$. $\gamma \circ \mathsf{Fstc}$ is the continuation for the seq-subcomputation $\langle \mathfrak{f}_{\mathrm{seq}} \mid \xi\rangle$ of $\langle \mathsf{fst}(\mathfrak{f}_{\mathrm{seq}}) \mid \xi\rangle$ with continuation γ, and $\gamma \circ \mathsf{Rstc}$ is the continuation for the seq-subcomputation of $\langle \mathsf{rst}(\mathfrak{f}_{\mathrm{seq}}) \mid \xi\rangle$ with continuation γ.

Continuation	Context for	Corresponding pfn[†]
Id	initial stage	
$\gamma \circ \langle \mathsf{Ifi}(\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}) \mid \xi\rangle$	test-subc of if-node	$\langle \lambda(s_{\mathrm{new}})\mathsf{if}(s_{\mathrm{new}},\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}) \mid \xi\rangle$
$\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle$	fun-subc of app-node	$\langle \lambda(s_{\mathrm{new}})\mathsf{app}(s_{\mathrm{new}},\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle$
$\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}})$	arg-subc of app-node	$\langle \lambda(\mathsf{x})\mathsf{app}(\mathsf{f},\mathsf{x}) \mid \mathsf{f} \leftarrow v_{\mathrm{fun}}\rangle$
$\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi\rangle$	lhs-subc of cart-node	$\langle \lambda(s_{\mathrm{new}})\mathsf{cart}(s_{\mathrm{new}},\mathfrak{f}_{\mathrm{rhs}}) \mid \xi\rangle$
$\gamma \circ \mathsf{Cartc}(v_{\mathrm{lhs}})$	rhs-subc of cart-node	$\langle \lambda(\mathsf{y})\mathsf{app}(\mathsf{x},\mathsf{y}) \mid \mathsf{x} \leftarrow v_{\mathrm{lhs}}\rangle$
$\gamma \circ \mathsf{Fstc}$	seq-subc of fst-node	$\lambda(\mathsf{x})\mathsf{fst}(\mathsf{x})$
$\gamma \circ \mathsf{Rstc}$	seq-subc of rst-node	$\lambda(\mathsf{x})\mathsf{rst}(\mathsf{x})$

[†] s_{new} is a symbol not free in any of the given forms

Figure 15. Continuations

The notation chosen for continuations reflects the view that a non-identity continuation is the composition of a continuation segment for the parent of the current node and the continuation representing the context of the parent node. Thus a continuation is essentially a sequence of continuation segments, one for each node along the path to the current node. We think of expressions such as $\langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle$ and $\mathsf{Appc}(v_{\mathrm{fun}})$ as denoting continuation segments and $\gamma \circ \boldsymbol{\theta}$ as the composition of the continuation γ and the continuation segment $\boldsymbol{\theta}$. A continuation segment for the parent of a node describes the remaining computation at the parent node as a function of the value returned at the child node (the subcomputation currently in progress).

Each continuation segment corresponds naturally to a pfn which is the abstraction of a dtree with respect to an immediate subcomputation. For example, abstracting the fun-subcomputation of an app-dtree, the pfn corresponding to $\langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle$ is $\langle \lambda(s_{\mathrm{new}})\mathsf{app}(s_{\mathrm{new}}, \mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle$. Abstracting the arg-subcomputation of an app-dtree where the fun- subcomputation has been completed, the pfn corresponding to $\mathsf{Appc}(v_{\mathrm{fun}})$ is $\langle \lambda(\mathsf{x})\mathsf{app}(\mathsf{f}, \mathsf{x}) \mid \mathsf{f} \leftarrow v_{\mathrm{fun}} \rangle$. The correspondence between continuation segments and pfns is also shown in Figure 15. Two continuations are equal iff they have the same number of segments and corresponding segments have equal corresponding pfns.

- **Concatenation of continuations.** Given the view of continuations as compositions, there is a natural notion of concatenation of continuations which is associative and for which Id is the left and right identity. We write $\gamma_0 \diamond \gamma_1$ for the concatenation of γ_0 and γ_1. For example,

$$(\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle) \diamond \mathsf{Id} = \gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle = \mathsf{Id} \diamond (\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle)$$

$$(\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle) \diamond (\mathsf{Id} \circ \mathsf{FstC}) = \gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle \circ \mathsf{FstC}$$

A continuation is concatenated (on the left) to a state by concatenation to the continuation component of the state.

$$\gamma_0 \diamond (\gamma \triangledown \mathfrak{d}) = (\gamma_0 \diamond \gamma) \triangledown \mathfrak{d} \quad \text{and} \quad \gamma_0 \diamond (\gamma \vartriangle v) = (\gamma_0 \diamond \gamma) \vartriangle v$$

A continuation is concatenated to a sequence of states by concatenating it to each element of the sequence. For example

$$\gamma \diamond [\varsigma_0, \varsigma_1, \varsigma_2] = [\gamma \diamond \varsigma_0, \gamma \diamond \varsigma_1, \gamma \diamond \varsigma_2]$$

V.1.4. Remarks

• **On the representation of stages as states.** Computation stages could serve directly as computation states, however we have chosen a more compact representation that corresponds more directly to the structures that might be used in mechanically carrying out sequential computations. The current node of a computation stage is exposed (accessed by single elementary operation). Completed subcomputation trees are pruned, leaving only the value. In the case of reduces-to, subcomputations are replaced by those to which they reduce. Thus continuations contain only the information about the context which is needed in order to continue the computation. This is a strong form of *tail-recursion*, in which "procedure call" (begin a subcomputation) is replaced by "goto" (reduce-to) whenever the result of the called procedure is to be returned as the result of the current computation.

Other choices for the representation of computation contexts may be appropriate for some purposes. For example, to provide the basis for an interpreter in an interactive system where the interpreter is expected to explain to the user what it has done at any stage of a computation (as a debugging tool) the entire computation tree context could be represented in the continuation. This would require only minor modifications in the step relation and in the theorems relating computation trees to computation sequences. In this case tail-recursiveness and hence some efficiency would be sacrificed in order to have more complete information available at each stage of a computation.

• **Determinism vs non-determinism.** An alternative to choosing a deterministic strategy for generating computation trees is to maintain the original rules for computation and to take continuations to be stages with a hole (a distinguished leaf with no label). This would result in a non-deterministic model of computation in which different choices for the next stage would in general give different results. We prefer to focus on the deterministic model for the present and to study properties of function and control abstraction in this mathematically simpler context. Non-determinism is introduced naturally when asynchronous computation primitives are introduced, for example, to model Actors (Hewitt [1977]). This seems like the appropriate framework in which to treat non-determinism.

V.2. Generating computation sequences

Figure 16 gives the rules for sequential computation. These rules define a single step relation ($\rightarrowtail_\iota$) on states, namely the least relation closed under the set of rules.

(sym) $\gamma \triangledown \langle s \mid \xi \rangle \rightarrowtail_\iota \gamma \vartriangle \xi(s)$

(dapp) $\gamma \triangledown o\,'\,d \rightarrowtail_\iota \gamma \vartriangle o(d)$

(lam) $\gamma \triangledown \langle \lambda(s)\mathfrak{f}_{\mathrm{body}} \mid \xi \rangle \rightarrowtail_\iota \gamma \vartriangle \langle \lambda(s)\mathfrak{f}_{\mathrm{body}} \mid \xi \rangle$

(app.fun) $\gamma \triangledown \langle \mathsf{app}(\mathfrak{f}_{\mathrm{fun}}, \mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle \rightarrowtail_\iota \gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle \triangledown \langle \mathfrak{f}_{\mathrm{fun}} \mid \xi \rangle$

(app.arg) $\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi \rangle \vartriangle v_{\mathrm{fun}} \rightarrowtail_\iota (\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}})) \triangledown \langle \mathfrak{f}_{\mathrm{fun}} \mid \xi \rangle$

(app.app) $\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}) \vartriangle v_{\mathrm{arg}} \rightarrowtail_\iota \gamma \triangledown (v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}})$

% if $v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}}$ is well-formed†

(if.test) $\gamma \triangledown \langle \mathsf{if}(\mathfrak{f}_{\mathrm{test}}, \mathfrak{f}_{\mathrm{then}}, \mathfrak{f}_{\mathrm{else}}) \mid \xi \rangle \rightarrowtail_\iota \gamma \circ \langle \mathsf{Ifi}(\mathfrak{f}_{\mathrm{then}}, \mathfrak{f}_{\mathrm{else}}) \mid \xi \rangle \triangledown \langle \mathfrak{f}_{\mathrm{test}} \mid \xi \rangle$

(if.br) $\gamma \circ \langle \mathsf{Ifi}(\mathfrak{f}_{\mathrm{then}}, \mathfrak{f}_{\mathrm{else}}) \mid \xi \rangle \vartriangle v_{\mathrm{test}} \rightarrowtail_\iota \gamma \triangledown \begin{cases} \langle \mathfrak{f}_{\mathrm{then}} \mid \xi \rangle & \text{if } v_{\mathrm{test}} \neq \square \\ \langle \mathfrak{f}_{\mathrm{else}} \mid \xi \rangle & \text{if } v_{\mathrm{test}} = \square \end{cases}$

(mt) $\gamma \triangledown \mathsf{mt} \rightarrowtail_\iota \gamma \vartriangle \square$

(cart.lhs) $\gamma \triangledown \langle \mathsf{cart}(\mathfrak{f}_{\mathrm{lhs}}, \mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle \rightarrowtail_\iota \gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle \triangledown \langle \mathfrak{f}_{\mathrm{lhs}} \mid \xi \rangle$

(cart.rhs) $\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi \rangle \vartriangle v_{\mathrm{lhs}} \rightarrowtail_\iota \gamma \circ \mathsf{Cartc}(v_{\mathrm{rhs}}) \triangledown \langle \mathfrak{f}_{\mathrm{rhs}} \mid \xi \rangle$

(cart.ret) $\gamma \circ \mathsf{Cartc}(v_{\mathrm{rhs}}) \vartriangle v_{\mathrm{rhs}} \rightarrowtail_\iota \gamma \vartriangle [v_{\mathrm{lhs}}, v_{\mathrm{rhs}}]$

(fst.seq) $\gamma \triangledown \langle \mathsf{fst}(\mathfrak{f}_{\mathrm{seq}}) \mid \xi \rangle \rightarrowtail_\iota \gamma \circ \mathsf{Fstc} \triangledown \langle \mathfrak{f}_{\mathrm{seq}} \mid \xi \rangle$

(fst.ret) $\gamma \circ \mathsf{Fstc} \vartriangle v_{\mathrm{seq}} \rightarrowtail_\iota \gamma \vartriangle 1^{\mathrm{st}}(v_{\mathrm{seq}})$

(rst.seq) $\gamma \triangledown \langle \mathsf{rst}(\mathfrak{f}_{\mathrm{seq}}) \mid \xi \rangle \rightarrowtail_\iota \gamma \circ \mathsf{Rstc} \triangledown \langle \mathfrak{f}_{\mathrm{seq}} \mid \xi \rangle$

(rst.ret) $\gamma \circ \mathsf{Rstc} \vartriangle v_{\mathrm{seq}} \rightarrowtail_\iota \gamma \vartriangle \mathrm{r}^{\mathrm{st}}(v_{\mathrm{seq}})$

(note) $\gamma \triangledown \langle \mathsf{note}(s)\mathfrak{f} \mid \xi \rangle \rightarrowtail_\iota \gamma \triangledown \langle \mathfrak{f} \mid \xi\{s \leftarrow \gamma\} \rangle$

(capp) $\gamma \triangledown \gamma_0\,'\,v \rightarrowtail_\iota \gamma_0 \vartriangle v$

† $v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}}$ is well-formed iff $v_{\mathrm{fun}} \in \mathbb{O} \wedge v_{\mathrm{arg}} \in \mathfrak{D}^*$ or $v_{\mathrm{fun}} \in \mathbb{P}$ or $v_{\mathrm{fun}} \in \mathbb{C}o$.

Figure 16. Rules for stepping

▪ Functionality of the single step relation. It is easy to see, since there is at most one step rule that applies to any computation state, that the single step relation is functional.

$$\varsigma \rightarrowtail_{\iota} \varsigma_0 \wedge \varsigma \rightarrowtail_{\iota} \varsigma_1 \rightarrow \varsigma_0 = \varsigma_1$$

The modified rules for generating computation trees translate naturally into step rules. If a computation stage is represented by ς_0 and the next stage is represented by ς_1 then $\varsigma_0 \rightarrowtail_{\iota} \varsigma_1$. For example, the step rule (app.fun) for beginning computation described by an app-dtree is derived from the begin rule (app.fun) (see §IV.2 Figure 11) using the construction of the continuation representing the context of the fun-subcomputation (given in §1).

(app.fun) $\gamma \triangledown \langle \mathsf{app}(\mathfrak{f}_{\text{fun}}, \mathfrak{f}_{\text{arg}}) \mid \xi \rangle \rightarrowtail_{\iota} \gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\text{arg}}) \mid \xi \rangle \triangledown \langle \mathfrak{f}_{\text{fun}} \mid \xi \rangle$

The step rule (app.arg) for returning the value of the fun-subcomputation is derived from the modified begin rule (app.arg)* given above using the construction of the context for the arg-subcomputation.

(app.arg) $\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\text{arg}}) \mid \xi \rangle \vartriangle v_{\text{fun}} \rightarrowtail_{\iota} \gamma \circ \mathsf{Appc}(v_{\text{fun}}) \triangledown \langle \mathfrak{f}_{\text{arg}} \mid \xi \rangle$

The step rule (app.app) for returning the value of the arg-subcomputation is derived from the reduces-to rule (app) combined with the begin rule (redt.beg).

(app.app) $\gamma \circ \mathsf{Appc}(v_{\text{fun}}) \vartriangle v_{\text{arg}} \rightarrowtail_{\iota} \gamma \triangledown v_{\text{fun}}{}' v_{\text{arg}}$

Note that the app-node disappears. Thus the return rule obtained by combining the rules (app) and (redt.ret) has no corresponding step rule.

The states not accounted for by this translation are those corresponding to noting and applying continuations. The step rule for beginning computation of $\langle \mathsf{note}(s_{\text{cont}})\mathfrak{f}_{\text{body}} \mid \xi \rangle$ with continuation γ says to bind γ to s_{cont} in ξ and begin computation of the body.

(note) $\gamma \triangledown \langle \mathsf{note}(s_{\text{cont}})\mathfrak{f}_{\text{body}} \mid \xi \rangle \rightarrowtail_{\iota} \gamma \triangledown \langle \mathfrak{f}_{\text{body}} \mid \xi\{s_{\text{cont}} \leftarrow \gamma\} \rangle$

The step rule for beginning computation of a continuation application dtree $\gamma_0{}' v$ says to return v to γ_0.

(capp) $\gamma \triangledown \gamma_0{}' v \rightarrowtail_{\iota} \gamma_0 \vartriangle v$

V.2.1. Computation sequences.

Now we define the notion of computation sequence and related notions useful for expressing properties of sequential computation.

▷ **Step Relation.** The step relation on computation states $\rightarrowtail$ is the transitive reflexive closure of the single step relation. $\rightarrowtail_\iota$. $\overset{+}{\rightarrowtail}$ is the transitive closure of $\rightarrowtail_\iota$.

$$\rightarrowtail \; = (\rightarrowtail_\iota)^* \quad \text{and} \quad \overset{+}{\rightarrowtail} = (\rightarrowtail_\iota)^+$$

▷ **Terminal states.** A *final* state is a return state with identity continuation, $\mathsf{Id} \vartriangle v$. A *hung* state is a state $\gamma \circ \mathsf{Appc}(v_{\text{fun}}) \vartriangle v_{\text{arg}}$ such that the application $v_{\text{fun}}{}' v_{\text{arg}}$ is not well-formed. A *terminal* state is a state that is either final or hung.

■ A computation state is terminal iff no step rule applies to it.

▷ **Computation sequence.** A non-empty (possibly infinite) sequence of states Σ is a computation sequence if for each state in the sequence, the next state is obtained by appling a step rule.

$$(\forall i < |\Sigma| - 1)(\Sigma{\downarrow}_i \rightarrowtail_\iota \Sigma{\downarrow}_{i+1})$$

We write $\varsigma_{\text{a}} \overset{\Sigma}{\rightarrowtail} \varsigma_{\text{z}}$ if Σ is a computation sequence of length $n+1$ with $\Sigma{\downarrow}_0 = \varsigma_{\text{a}}$ and $\Sigma{\downarrow}_n = \varsigma_{\text{z}}$.

▷ **The computation sequence for a state or dtree.** Using the functionality of the single step relation, we define $\Sigma(\varsigma)$, the computation sequence from the state ς, to be the limit of the computation sequences beginning with ς. Thus $\Sigma(\varsigma)$ is the unique computation sequence which has as an initial segment each computation sequence beginning with ς. The computation sequence $\Sigma(\partial)$ for a dtree ∂ is defined by $\Sigma(\partial) \underset{\text{df}}{=} \Sigma(\mathsf{Id} \triangledown \partial)$.

Figure 17 shows the computation sequence for ∂_{if} (defined in §IV.2). The dtrees in the left margin mark the beginning ($\triangledown\ \partial$) and ($\vartriangle\ \partial$) end of the subsequences corresponding to the subcomputation described by ∂. This shows the preservation of the tree structure in the computation sequence (see Figure 13 in §IV.2). Note that returning a value along any chain of reduces-to subcomputations of a computation tree is accomplished in a single step in a computation sequence.

$$\begin{array}{ll}
\triangledown\, \partial_{\mathrm{if}} & \mathrm{Id} \triangledown \langle \mathrm{if}(\mathrm{z}, \mathfrak{f}_{\mathrm{cart}}, \mathrm{mt}) \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\quad \triangledown\, \partial_{\mathrm{z}} & \mathrm{Id} \circ \langle \mathrm{Ifi}(\mathfrak{f}_{\mathrm{cart}}, \mathrm{mt}) \mid \mathrm{z} \leftarrow v \rangle \triangledown \langle \mathrm{z} \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\quad \vartriangle \partial_{\mathrm{z}} & \mathrm{Id} \circ \langle \mathrm{Ifi}(\mathfrak{f}_{\mathrm{cart}}, \mathrm{mt}) \mid \mathrm{z} \leftarrow v \rangle \vartriangle v \\
 & \rightarrowtail_{\iota} \\
\quad \triangledown\, \partial_{\mathrm{cart}} & \mathrm{Id} \triangledown \langle \mathrm{cart}(\mathfrak{f}_{\mathrm{fst}}, \mathfrak{f}_{\mathrm{app}}) \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad \triangledown\, \partial_{\mathrm{fst}} & \mathrm{Id} \circ \langle \mathrm{Carti}(\mathfrak{f}_{\mathrm{app}}) \mid \mathrm{z} \leftarrow v \rangle \triangledown \langle \mathrm{fst}(\mathrm{z}) \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \triangledown\, \partial_{\mathrm{z}} & \mathrm{Id} \circ \langle \mathrm{Carti}(\mathfrak{f}_{\mathrm{app}}) \mid \mathrm{z} \leftarrow v \rangle \circ \mathrm{Fstc} \triangledown \langle \mathrm{z} \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \vartriangle \partial_{\mathrm{z}} & \mathrm{Id} \circ \langle \mathrm{Carti}(\mathfrak{f}_{\mathrm{app}}) \mid \mathrm{z} \leftarrow v \rangle \circ \mathrm{Fstc} \vartriangle v \\
 & \rightarrowtail_{\iota} \\
\qquad \vartriangle \partial_{\mathrm{fst}} & \mathrm{Id} \circ \langle \mathrm{Carti}(\mathfrak{f}_{\mathrm{app}}) \mid \mathrm{z} \leftarrow v \rangle \vartriangle a \\
 & \rightarrowtail_{\iota} \\
\qquad \triangledown\, \partial_{\mathrm{app}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \triangledown \langle \mathrm{app}(\mathfrak{f}_{\mathrm{k}}, \mathfrak{f}_{\mathrm{rst}}) \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \triangledown\, \partial_{\mathrm{k}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \langle \mathrm{Appi}(\mathfrak{f}_{\mathrm{rst}}) \mid \mathrm{z} \leftarrow v \rangle \triangledown \lambda(\mathrm{x})\lambda(\mathrm{y})\mathrm{x} \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \vartriangle \partial_{\mathrm{k}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \langle \mathrm{Appi}(\mathfrak{f}_{\mathrm{rst}}) \mid \mathrm{z} \leftarrow v \rangle \vartriangle \lambda(\mathrm{x})\lambda(\mathrm{y})\mathrm{x} \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \triangledown\, \partial_{\mathrm{rst}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \mathrm{Appc}(\mathrm{K}) \triangledown \langle \mathrm{rst}(\mathrm{z}) \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad\qquad \triangledown\, \partial_{\mathrm{z}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \mathrm{Appc}(\mathrm{K}) \circ \mathrm{RstC} \triangledown \langle \mathrm{z} \mid \mathrm{z} \leftarrow v \rangle \\
 & \rightarrowtail_{\iota} \\
\qquad\qquad \vartriangle \partial_{\mathrm{z}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \mathrm{Appc}(\mathrm{K}) \circ \mathrm{RstC} \vartriangle v \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \vartriangle \partial_{\mathrm{rst}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \circ \mathrm{Appc}(\mathrm{K}) \vartriangle u \\
 & \rightarrowtail_{\iota} \\
\qquad\quad \triangledown\, \partial_{\mathrm{ku}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \triangledown \mathrm{K}'u \\
 & \rightarrowtail_{\iota} \\
\qquad \vartriangle \partial_{\mathrm{app}} \vartriangle \partial_{\mathrm{ku}} & \mathrm{Id} \circ \mathrm{Cartc}(a) \vartriangle \mathrm{K}(u) \\
 & \rightarrowtail_{\iota} \\
\vartriangle \partial_{\mathrm{if}} \vartriangle \partial_{\mathrm{cart}} & \mathrm{Id} \vartriangle [a, \mathrm{K}(u)]
\end{array}$$

where

$$\begin{array}{lll}
\mathfrak{f}_{\mathrm{cart}} = \mathrm{cart}(\mathfrak{f}_{\mathrm{fst}}, \mathfrak{f}_{\mathrm{app}}) & \mathfrak{f}_{\mathrm{fst}} = \mathrm{fst}(\mathrm{z}) & v = [a, u] \\
\mathfrak{f}_{\mathrm{app}} = \mathrm{app}(\mathfrak{f}_{\mathrm{k}}, \mathfrak{f}_{\mathrm{rst}}) & \mathfrak{f}_{\mathrm{rst}} = \mathrm{rst}(\mathrm{z}) & \mathfrak{f}_{\mathrm{k}} = \lambda(\mathrm{x})\lambda(\mathrm{y})\mathrm{x}
\end{array}$$

Figure 17. Computation sequence for ∂_{if}

V.3. Context insensitivity and context independence

Although the computation described by a dtree in general depends on the context in which the computation is carried out, this dependence is uniformly determined by the dtree and in many cases a dtree or pfn has context independent behavior, although continuations may be noted or applied within the computation sequence. In this section we introduce some additional dtree properties in order to express the "context insensitivity" of dtrees and to identify a fragment of s-*Rum* in which tools for reasoning about context independent computation can be used.

V.3.1. Classification of dtrees

We begin by defining a *sequential equivalence* relation ($\approx$) on dtrees of s-*Rum* analogous to the ctree equivalence relation on dtrees of t-*Rum*.

▹ **Sequential equivalence.** Two dtrees are equivalent with respect to sequential computation (abbreviated c-seq equivalent) iff in any given context, if computation described by one dtree reaches a final state then computation described by the other dtree reaches the same final state.

$$\mathfrak{d}_0 \approx \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} (\forall \gamma, v)(\gamma \triangledown \mathfrak{d}_0 \rightarrowtail \mathsf{Id} \vartriangle v \leftrightarrow \gamma \triangledown \mathfrak{d}_1 \rightarrowtail \mathsf{Id} \vartriangle v)$$

By context insensitivity we mean that the computation described by a dtree either returns a value to the calling continuation, escapes to the top level, or diverges. Which case holds depends only on the dtree, not on the context in which the computation is carried out. This is made precise by the following definitions and the dtree classification theorem.

▹ **Return to caller.** $\mathfrak{d}$ returns a value to the calling continuation (${\downarrow}\mathfrak{d}$) is defined by

$${\downarrow}\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\forall \gamma)(\exists v)\gamma \triangledown \mathfrak{d} \rightarrowtail \gamma \vartriangle v$$

▹ **Escape to top.** $\mathfrak{d}$ escapes to the top level (${}^{\mathsf{Id}}{\downarrow}\mathfrak{d}$) is defined by

$${}^{\mathsf{Id}}{\downarrow}\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\forall \gamma)(\exists v)\gamma \triangledown \mathfrak{d} \rightarrowtail \mathsf{Id} \vartriangle v$$

▹ **Diverge.** $\mathfrak{d}$ diverges (${\uparrow}\mathfrak{d}$) is defined by

$${\uparrow}\mathfrak{d} \underset{\mathrm{df}}{\equiv} \neg(\exists v)(\mathsf{Id} \triangledown \mathfrak{d} \rightarrowtail \mathsf{Id} \vartriangle v)$$

■ **Dtree classification theorem.** The three classes of dtrees defined above are a partition of $\mathbb{D}t$.

$$\downarrow\mathfrak{d} \vee {}^{\mathsf{Id}}\!\downarrow\mathfrak{d} \vee \uparrow\mathfrak{d} \quad \text{and} \quad (\downarrow\mathfrak{d} \to \neg{}^{\mathsf{Id}}\!\downarrow\mathfrak{d}) \wedge ({}^{\mathsf{Id}}\!\downarrow\mathfrak{d} \to \neg\uparrow\mathfrak{d}) \wedge (\uparrow\mathfrak{d} \to \neg\downarrow\mathfrak{d})$$

This will be proved at the end of this section. Some additional simple facts are

■ If $\mathfrak{d}$ diverges then either $\Sigma(\mathfrak{d})$ is infinite or $\Sigma(\mathfrak{d})$ terminates in a hung state. Thus if $\mathfrak{d}$ diverges the computation described by $\mathfrak{d}$ in any context never reaches a final state.

$$\uparrow\mathfrak{d} \to \neg(\exists\gamma, v)(\gamma \triangledown \mathfrak{d} \rightarrowtail \mathsf{Id} \vartriangle v)$$

■ Sequential equivalence respects the classification of dtrees – if two dtrees are c-seq equivalent then they belong to the same classification.

$$\mathfrak{d}_0 \approx \mathfrak{d}_1 \to \downarrow\mathfrak{d}_0 \leftrightarrow \downarrow\mathfrak{d}_1 \wedge {}^{\mathsf{Id}}\!\downarrow\mathfrak{d}_0 \leftrightarrow {}^{\mathsf{Id}}\!\downarrow\mathfrak{d}_1 \wedge \uparrow\mathfrak{d}_0 \leftrightarrow \uparrow\mathfrak{d}_1$$

Remark. A dtree that always returns a value to the calling continuation may not always return the same value. For example the dtree note(g)g returns the calling continuation to the calling continuation – $\gamma \triangledown \mathsf{note(g)g} \rightarrowtail \gamma \vartriangle \gamma$. However the proof of the dtree classification theorem will also show that the value returned is itself uniformly parameterized by the calling continuation.

V.3.2. Preservation of tree structure in sequential computation

The close correspondence between the structure of the computation tree for $\mathfrak{d}_{\text{if}}$ (Figure 13 in §IV.2) and the computation sequence for $\mathfrak{d}_{\text{if}}$ (Figure 17 in §2) holds for dtrees of t-$\mathcal{R}\mathit{um}$ in general. The next three facts express the sense in which the structure of computation trees is preserved in computation sequences for t-$\mathcal{R}\mathit{um}$ dtrees.

■ **Preservation of evaluation.** A dtree $\mathfrak{d}$ of t-$\mathcal{R}\mathit{um}$ has value v iff the computation sequence for $\mathfrak{d}$ reaches a final state with the value v iff $\mathfrak{d}$ returns v in any context.

$$\mathfrak{d} \in \text{t-}\mathbb{D}t \to \mathfrak{d} \hookrightarrow v \leftrightarrow \mathsf{Id} \triangledown \mathfrak{d} \overset{\Sigma(\mathfrak{d})}{\rightarrowtail} \mathsf{Id} \vartriangle v \leftrightarrow (\forall\gamma)\gamma \triangledown \mathfrak{d} \overset{\gamma\circ\Sigma(\mathfrak{d})}{\rightarrowtail} \gamma \vartriangle v$$

■ **Dtrees of t-$\mathcal{R}\mathit{um}$ can't escape.** The sequential computation described by a dtree $\mathfrak{d}$ of t-$\mathcal{R}\mathit{um}$ must return a value to the caller or diverge. It cannot escape.

$$\mathfrak{d} \in \text{t-}\mathbb{D}t \to \downarrow\mathfrak{d} \vee \uparrow\mathfrak{d}$$

▪ **Preservation of reduces-to.** For t-*Rum* dtrees $\mathfrak{d}_0$ reduces-to $\mathfrak{d}_1$ iff there is a non-trivial computation sequence from $\mathfrak{d}_0$ to $\mathfrak{d}_1$.

$$\mathfrak{d}_0 \in \text{t-}\mathbb{D}t \rightarrow (\mathfrak{d}_0 \mathrel{\gg\!\!-} \mathfrak{d}_1 \leftrightarrow (\forall\gamma)(\gamma \triangledown \mathfrak{d}_0 \overset{+}{\rightarrowtail} \gamma \triangledown \mathfrak{d}_1))$$

▪ **Preservation of subcomputation.** For a defined dtree of t-*Rum* each subcomputation corresponds to a subsequence of the computation sequence.

$$\mathfrak{d} \in \text{t-}\mathbb{D}t \wedge \mathfrak{d} \hookrightarrow v \wedge \mathfrak{d}_0 \prec \mathfrak{d} \rightarrow (\exists\gamma, \Sigma_0, \Sigma_1)(\Sigma[\mathfrak{d}] = [\Sigma_0, \gamma \diamond \Sigma[\mathfrak{d}_0], \Sigma_1])$$

In the case $\mathfrak{d} \mathrel{\gg\!\!-} \mathfrak{d}_0$ Σ_1 will in fact be the empty sequence. Otherwise it will be a proper computation sequence.

V.3.3. Context independence

We extend the notions of evaluation, reduces-to, definedness, and ctree equivalence to s-*Rum* and introduce the notion of local dtree ($\uparrow\!\downarrow\mathfrak{d}$). The computation described by a local dtree either diverges or returns the *same* value to any calling context. The notion of subcomputation does not in general make sense in s-*Rum*, even for local dtrees, so we do not attempt to extend this notion.

▹ **Evaluation.** $\mathfrak{d} \hookrightarrow v \underset{\mathrm{df}}{\equiv} (\forall\gamma)(\gamma \triangledown \mathfrak{d} \rightarrowtail \gamma \vartriangle v)$

▹ **Reduces-to.** $\mathfrak{d}_0 \mathrel{\gg\!\!-} \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} (\forall\gamma)(\gamma \triangledown \mathfrak{d}_0 \overset{+}{\rightarrowtail} \gamma \triangledown \mathfrak{d}_1)$

▹ **Definedness.** $\Downarrow\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\exists v)\mathfrak{d} \hookrightarrow v$

▹ **Local.** $\uparrow\!\downarrow\mathfrak{d} \underset{\mathrm{df}}{\equiv} \Downarrow\mathfrak{d} \vee \uparrow\mathfrak{d}$

▹ **Ctree equivalence.** $\mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} \uparrow\!\downarrow\mathfrak{d}_0 \wedge \uparrow\!\downarrow\mathfrak{d}_1 \wedge (\forall v)(\mathfrak{d}_0 \hookrightarrow v \leftrightarrow \mathfrak{d}_1 \hookrightarrow v)$

▹ **Pfnl.** $\varphi \in \mathit{Pfnl} \underset{\mathrm{df}}{\equiv} (\forall v)(\exists\varphi_1)\varphi' v \hookrightarrow \varphi_1$

Note that the definitions of definedness and pfnl have the same form as before. It follows easily from the facts expressing the preservation of tree structure in sequential computation that the new definitions of $\hookrightarrow$, $\prec$, $\gg\!\!-$, $\Downarrow$, and $\rightleftharpoons$ agree with the original definitions when restricted to t-*Rum*.

Some simple properties of locality and the computation relations are expressed by the following facts which follow easily from the definitions

▪ Dtrees of t-*Rum* are local.

$$\mathfrak{d} \in \text{t-}\mathit{Rum} \rightarrow \uparrow\!\downarrow\mathfrak{d}$$

- If $\mathfrak{d}$ is defined then $\mathfrak{d}$ returns a value (the same value) to any calling continuation.

$$\Downarrow\mathfrak{d} \to \downarrow\mathfrak{d}$$

The note(g)g example shows that the converse is not true.

$$\downarrow\text{note(g)g} \quad \text{and} \quad \neg\Downarrow\text{note(g)g}$$

- If $\mathfrak{d}_0$ reduces-to $\mathfrak{d}_1$ then $\mathfrak{d}_0$ is local iff $\mathfrak{d}_1$ is local.

$$\mathfrak{d}_0 \succ\!\!\succ \mathfrak{d}_1 \to (\uparrow\!\downarrow\mathfrak{d}_0 \leftrightarrow \uparrow\!\downarrow\mathfrak{d}_1)$$

- If $\mathfrak{d}_0$ reduces-to $\mathfrak{d}_1$ then $\mathfrak{d}_0$ and $\mathfrak{d}_1$ are c-seq equivalent.

$$\mathfrak{d}_0 \succ\!\!\succ \mathfrak{d}_1 \to \mathfrak{d}_0 \approx \mathfrak{d}_1$$

- If $\mathfrak{d}_0$ are ctree equivalent $\mathfrak{d}_1$ then $\mathfrak{d}_0$ and $\mathfrak{d}_1$ are c-seq equivalent.

$$\mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1 \to \mathfrak{d}_0 \approx \mathfrak{d}_1$$

- For local dtrees reduces-to implies ctree equivalence.

$$\mathfrak{d}_0 \succ\!\!\succ \mathfrak{d}_1 \land \uparrow\!\downarrow\mathfrak{d}_0 \to \mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1$$

V.3.4. Lifting theorems from t-*Rum*

Local dtrees have many of the properties of t-*Rum* dtrees. This allows us to lift many of the basic theorems of t-*Rum* to s-*Rum*. The following are some examples.

- The computation rules hold in *Rum* for the extended definitions of evaluation and reduces-to. For example,

(redt.ret) $\quad \mathfrak{d} \succ\!\!\succ \mathfrak{d}_0 \land \mathfrak{d}_0 \hookrightarrow v \to \mathfrak{d} \hookrightarrow v$

(if) $\quad \langle \mathfrak{f}_{\text{test}} \mid \xi \rangle \hookrightarrow u \to \langle \text{if}(\mathfrak{f}_{\text{test}}, \mathfrak{f}_{\text{then}}, \mathfrak{f}_{\text{else}}) \mid \xi \rangle \succ\!\!\succ \begin{cases} \langle \mathfrak{f}_{\text{then}} \mid \xi \rangle & \text{if } u \neq \square \\ \langle \mathfrak{f}_{\text{else}} \mid \xi \rangle & \text{if } u = \square \end{cases}$

(app) $\quad \langle \mathfrak{f}_{\text{fun}} \mid \xi \rangle \hookrightarrow v_{\text{fun}} \land \langle \mathfrak{f}_{\text{arg}} \mid \xi \rangle \hookrightarrow v_{\text{arg}} \to \langle \mathfrak{f}_{\text{fun}}(\mathfrak{f}_{\text{arg}}) \mid \xi \rangle \succ\!\!\succ v_{\text{fun}}{}' v_{\text{arg}}$

% if $v_{\text{fun}}{}' v_{\text{arg}}$ is well-formed

Since the notion of subcomputation does not extend usefully to s-$\mathcal{R}um$ computation induction as formulated in §IV.2 is limited to t-$\mathcal{R}um$. However, if subcomputation is replaced by reduces-to the computation induction formula remains valid and is a useful special case.

- For any dtree property Q

$$\text{(s.CI)} \qquad (\forall\mathfrak{d})((\forall\mathfrak{d}_0)(\mathfrak{d} \gg\!\!- \mathfrak{d}_0 \to Q(\mathfrak{d}_0)) \to Q(\mathfrak{d})) \to (\forall\mathfrak{d})(\Downarrow\mathfrak{d} \to Q(\mathfrak{d}))$$

- The reduces-to laws (I.redt), (K.redt), (Cl.redt), (B.redt), and (S.redt) given for the algebraic combinator pfns in §IV.4 hold in s-$\mathcal{R}um$.
- For any pfnl ϑ the following properties of the recursion pfn hold in s-$\mathcal{R}um$

(rec.pfnl) $\mathsf{Rec} \in \boldsymbol{Pfnl}$

(rec.redt) $\mathsf{Rec}(\vartheta)'v \gg\!\!- \vartheta(\mathsf{Rec}(\vartheta))'v$

(rec.fix.s) $\mathsf{Rec}(\vartheta)'v \approx \vartheta(\mathsf{Rec}(\vartheta))'v$

(rec.subc) is omitted as it only makes sense in t-$\mathcal{R}um$.

In s-$\mathcal{R}um$ the properties stated for the sequence recursion pfnls hold when restricted to the local cases.

- Collect and Tuple satisfy

$$\uparrow\!\downarrow\vartheta'a_1, \wedge \ldots, \wedge \uparrow\!\downarrow\vartheta'a_{\mathrm{n}} \to \mathsf{Collect}(\vartheta)'[a_1,\ldots,a_{\mathrm{n}}] \rightleftharpoons [\vartheta'a_1,\ldots,\vartheta'a_{\mathrm{n}}]$$

and

$$\uparrow\!\downarrow\vartheta_1'a, \wedge \ldots, \wedge \uparrow\!\downarrow\vartheta_{\mathrm{n}}'a \to \mathsf{Tuple}[\vartheta_1,\ldots,\vartheta_{\mathrm{n}}]'u \rightleftharpoons [\vartheta_1'u,\ldots,\vartheta_{\mathrm{n}}'u]$$

- The characterization of Some is valid in s-$\mathcal{R}um$ as stated.
- For pfns ϑ and domains A, U such that $A \subset \mathbf{V}$ and $U \subset \mathbf{V}^*$, Seqlt satisfies

$$(\forall a \in A)(\forall u \in U)(\exists v \in U)(\vartheta'(a,u) \hookrightarrow v)$$

$$\to$$

$$(\forall\{a_1, \ldots\ a_{\mathrm{m}}\} \subset A)(\forall u \in U)$$

$$\mathsf{Seqlt}'(\vartheta, u, [a_1, \ldots a_{\mathrm{m}}]) \rightleftharpoons \vartheta'(a_1, \ldots, \vartheta'(a_{\mathrm{m}}, u)\ldots)$$

V.3.5. Proof of the dtree classification theorem

We outline the proof of the dtree classification theorem as it uses a trick which may have useful applications in other situations. The trick is to work with objects parameterized by a single continuation parameter and to analyze the properties of parameterized computation sequences. The continuation parameter provides a way of tagging a continuation and identifying continuations within a computation sequence that are generated from a tagged continuation. For the non-divergent cases we will prove stronger statements of the form "there exists a parameterized value such that for all continuations ..." rather than "for all continuations there exists a parameterized value such that ...".

Parameterized objects are generated just as the objects of s-*Rum* except that in addition to the primitive continuation Id, there is a continuation parameter #. We will refer to these objects as p-*Rum* objects and we use the same variables to range over the various sorts of p-*Rum* objects as for t-*Rum* and s-*Rum*. Note that the s-*Rum* objects are a subset of the p-*Rum* objects. For a parameterized object χ and a continuation γ we write $\chi\{\gamma\}$ for the object obtained by replacing all occurrences of # in χ by γ.[1] If γ is an s-*Rum* continuation then $\chi\{\gamma\}$ is an s-*Rum* object. The rules for sequential computation have exactly the same form for p-*Rum* as for s-*Rum*. In p-*Rum* there is a new kind of terminal computation state $\# \vartriangle v$ – which corresponds to *return to caller.*

The following facts are easy to establish:

■ **Substitution lemma.** The result of replacing the parameter in a computation sequence by a continuation γ is a computation sequence.

$$\varsigma_0 \overset{\Sigma}{\rightarrowtail} \varsigma_1 \rightarrow \varsigma_0\{\gamma\} \overset{\Sigma\{\gamma\}}{\rightarrowtail} \varsigma_1\{\gamma\}$$

[1] This is a use of braces to express a form of substitution. Context will readily distinguish this use from other uses such as for readability or Kleene braces.

■ **Classification of parameterized states.** For any parameterized state ς exactly one of the following holds:

(i) The computation sequence beginning with ς terminates with a state $\# \vartriangle v$ for some v.

$$(\exists v)\varsigma \rightarrowtail \# \vartriangle v$$

(ii) The computation sequence beginning with ς terminates with a state $\mathsf{Id} \vartriangle v$ for some v.

$$(\exists v)\varsigma \rightarrowtail \mathsf{Id} \vartriangle v$$

(iii) The computation sequence beginning with ς hangs.

$$(\exists \gamma, v_{\mathrm{fun}} v_{\mathrm{arg}})(v_{\mathrm{fun}}{}' v_{\mathrm{arg}} \text{ not well-formed} \wedge \varsigma \rightarrowtail \gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}) \vartriangle v_{\mathrm{arg}})$$

(iv) The computation sequence beginning with ς is infinite.

Note that in (i), (ii), and (iii) the values will in general be parameterized. Taking ς to be $\# \triangledown \eth$, collapsing the latter two cases into the divergent case, and using the substitution lemma we have

■ **Uniform dtree classification.** For $\eth$ and γ in s-*Rum* exactly one of the following holds

(i) $(\exists v)(\forall \gamma)\gamma \triangledown \eth \rightarrowtail \gamma \vartriangle v\{\gamma\}$

(ii) $(\exists v)(\forall \gamma)\gamma \triangledown \eth \rightarrowtail \mathsf{Id} \vartriangle v\{\gamma\}$

(iii) $\neg(\exists v)(\mathsf{Id} \triangledown \eth \rightarrowtail \mathsf{Id} \vartriangle v)$

The dtree classification theorem now follows easily.

V.4. Looping and escaping

As our first example of programming and proving with continuations, we prove the equivalence of two pfnls Until and DoUntil that construct an *until loop* from a test and an action. The until loop is typical of the looping constructs used in programming. Given a test $\vartheta_{\mathrm{test}}$ and an action ϑ_{act}, the until loop for $(\vartheta_{\mathrm{test}}, \vartheta_{\mathrm{act}})$ repeatedly applies ϑ_{act} to an argument u until u satisfies $\vartheta_{\mathrm{test}}$, at which point u is returned as the result. Until is a t-*Rum* pfn defined by a simple recursive definition.

▷ $\mathsf{Until} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{un})\lambda(\mathsf{s})\mathsf{if}(\mathsf{p}(\mathsf{s}),\mathsf{s},\mathsf{un}(\mathsf{f}(\mathsf{s}))))$

■ **Until lemma.** The key computational properties of Until are

$$\text{(until.done)}\quad \vartheta_{\text{test}}{}'u \hookrightarrow [a,v] \;\to\; \mathsf{Until}(\vartheta_{\text{test}},\vartheta_{\text{act}})'u \hookrightarrow u$$

$$\text{(until.loop)}\quad \vartheta_{\text{test}}{}'u \hookrightarrow \Box \wedge \vartheta_{\text{act}}{}'u \hookrightarrow v \;\to$$
$$\mathsf{Until}(\vartheta_{\text{test}},\vartheta_{\text{act}})'u \mathrel{\succ\!\!\succ} \mathsf{Until}(\vartheta_{\text{test}},\vartheta_{\text{act}})'v$$

DoUntil is based on the *do-forever loop.* A do-forever loop has a pfn parameter which is repeatedly applied to the loop argument. The pfn Do describes a do-forever loop. DoUntil notes the calling continuation and constructs a do-forever loop with a pfn parameter that contains the test pfn, the action pfn, and the calling continuation. This pfn applies the test pfn to the loop argument and then applies the calling continuation (to return to the caller) or it applies the action pfn (to continue looping) according to whether the test succeeds or fails. DoU is a variant of DoUntil used to formulate properties of DoUntil. DoU has an additional parameter intended to be the calling continuation.

$$\triangleright\qquad \mathsf{Do} \overset{\text{df}}{\longleftrightarrow} \lambda(\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{Do})\lambda(\mathsf{z})\mathsf{Do}(\mathsf{f}(\mathsf{z})))$$

$$\triangleright\qquad \mathsf{DoUntil} \overset{\text{df}}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f})\lambda(\mathsf{z})\mathsf{note}(\mathsf{g})\mathsf{Do}(\lambda(\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{g}(\mathsf{x}),\mathsf{f}(\mathsf{x})),\mathsf{z})$$

$$\triangleright\qquad \mathsf{DoU} \overset{\text{df}}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f},\mathsf{g})\lambda(\mathsf{z})\mathsf{Do}(\lambda(\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{g}(\mathsf{x}),\mathsf{f}(\mathsf{x})),\mathsf{z})$$

■ **DoUntil lemma.** The key computational facts about DoUntil are

$$\text{(do.note)}\quad \gamma \triangledown \mathsf{DoUntil}(\vartheta_{\text{test}},\vartheta_{\text{act}})'v \rightarrowtail \gamma \triangledown \mathsf{DoU}(\vartheta_{\text{test}},\vartheta_{\text{act}},\gamma)'v$$

$$\text{(do.done)}\quad \vartheta_{\text{test}}{}'v \hookrightarrow [a,u] \;\to\; \mathsf{DoU}(\vartheta_{\text{test}},\vartheta_{\text{act}},\gamma)'v \approx \gamma'v$$

$$\text{(do.loop)}\quad \vartheta_{\text{test}}{}'v \hookrightarrow \Box \wedge \vartheta_{\text{act}}{}'v \hookrightarrow u \;\to$$
$$\mathsf{DoU}(\vartheta_{\text{test}},\vartheta_{\text{act}},\gamma)'v \mathrel{\succ\!\!\succ} \mathsf{DoU}(\vartheta_{\text{test}},\vartheta_{\text{act}},\gamma)'u$$

From the basic facts about Until and DoUntil (and computation induction) it is clear that they describe essentially the same computations. More precisely we have the following equivalences.

■ **t-*Rum* equivalence of Until and DoUntil.** For ϑ_{test}, ϑ_{act} and v in t-*Rum*

$$\mathsf{DoUntil}(\vartheta_{\text{test}},\vartheta_{\text{act}})'v \rightleftharpoons \mathsf{Until}(\vartheta_{\text{test}},\vartheta_{\text{act}})'v$$

■ **s-*Rum* equivalence of Until and DoUntil.** For $U \subset \mathbf{V}^*$

$$(\forall u \in U)(\Downarrow\vartheta_{\text{test}}{}'u \wedge (\exists v \in U)\vartheta_{\text{act}}{}'u \hookrightarrow v) \;\to$$
$$(\forall u \in U)(\mathsf{DoUntil}(\vartheta_{\text{test}},\vartheta_{\text{act}})'u \rightleftharpoons \mathsf{Until}(\vartheta_{\text{test}},\vartheta_{\text{act}})'u)$$

Remarks

- Note that $\mathsf{Do}(\vartheta)$ is everywhere undefined for any ϑ in t-*Rum*.

- We have now provided all the tools needed to interpret and verify the statements about sequential computation in the tree product examples (Chapter II). In particular the equations for Tprod2 and the derivation of Tprodg can now be verified. We leave this as an exercise for the reader.

Exercise 1. Proving equations with continuations: define Find, FFind as follows

$$\triangleright \qquad \mathsf{Find} \overset{df}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{x})\mathsf{note}(\mathsf{g})\mathsf{Finder}(\mathsf{p},\mathsf{g})(\mathsf{x})$$

$$\triangleright \qquad \mathsf{Finder} \overset{df}{\longleftrightarrow} \begin{aligned}[t] &\lambda(\mathsf{p},\mathsf{g})\mathsf{Rec}(\lambda(\mathsf{fnd})\lambda(\mathsf{x})\\ &\qquad\qquad \mathsf{if}(\mathsf{p}(\mathsf{x}),\\ &\qquad\qquad\quad \mathsf{g}(\mathsf{x}),\\ &\qquad\qquad\quad \mathsf{if}(\mathsf{Atom}(\mathsf{x}),\mathsf{mt},[\mathsf{fnd}(\mathsf{Car}(\mathsf{x})),\mathsf{fnd}(\mathsf{Cdr}(\mathsf{x}))])))\end{aligned}$$

$$\triangleright \qquad \mathsf{FFind} \overset{df}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f},\mathsf{x})\mathsf{note}(\mathsf{g})\mathsf{FFinder}(\mathsf{p},\mathsf{f},\mathsf{g})(\mathsf{x})$$

$$\triangleright \qquad \mathsf{FFinder} \overset{df}{\longleftrightarrow} \begin{aligned}[t] &\lambda(\mathsf{p},\mathsf{f},\mathsf{g})\mathsf{Rec}(\lambda(\mathsf{ffnd})\lambda(\mathsf{x})\\ &\qquad\qquad \mathsf{if}(\mathsf{p}(\mathsf{x}),\\ &\qquad\qquad\quad \mathsf{g}(\mathsf{f}(\mathsf{x})),\\ &\qquad\qquad\quad \mathsf{if}(\mathsf{Atom}(\mathsf{x}),\mathsf{mt},[\mathsf{ffnd}(\mathsf{Car}(\mathsf{x})),\mathsf{ffnd}(\mathsf{Cdr}(\mathsf{x}))])))\end{aligned}$$

Prove that the following equation holds for x ranging over S-expressions and f and p ranging over pfns which are local on S-expressions.

$$\mathsf{Ffind}(\mathsf{p},\mathsf{f},\mathsf{x}) \rightleftharpoons \mathsf{let}\{\mathsf{y} \leftarrow \mathsf{Find}(\mathsf{p},\mathsf{x})\}\mathsf{if}(\mathsf{y},\mathsf{f}(\mathsf{y}),\mathsf{mt})$$

V.5. Co-routines in *Rum*

As a further example of programming with continuations, we show how the co-routine mechanism can be represented in *Rum*. Recall (§I.2) that a system of co-routines is a set of programs that interact by resuming one another rather than by the usual function calling (application) mechanism. A given co-routine when resumed will continue computation where it last left off, carry out the next portion of its computation, and then resume some other co-routine. Typically information is passed between co-routines by updating shared data structures and each co-routine has internal state that keeps track of where to begin when it is resumed. In *Rum* the shared data and resumption points must be passed explicitly as parameters and a co-routine is continuation satisfying certain conditions. Here we consider co-routines characterized by the sequences they generate, and to avoid irrelevant details we will assume the sequences are infinite, i.e. functions on the natural numbers. For simplicity in discussing co-routines we define a step relation on continuation application dtrees.

$$\gamma_0{}'v_0 \rightarrowtail \gamma_1{}'v_1 \underset{\mathrm{df}}{=} (\forall\gamma)(\exists\gamma_2)(\gamma \triangledown \gamma_0{}'v_0 \rightarrowtail \gamma_2 \triangledown \gamma_1{}'v_1$$

▷ **Co-routine generating a sequence.** Let σ be a sequence of elements from the computation domain. We say that a continuation γ_{co} is a co-routine generating the sequence σ if there is a sequence of continuations Γ such that

$$\Gamma(0) = \gamma_{\mathrm{co}} \quad \text{and} \quad \Gamma(i)'\gamma \rightarrowtail \gamma'[\Gamma(i+1), \sigma(i)]$$

for any γ and number i.

Co-routine resumption is based on continuation application. The pfn Resume describes resumption for the case of simple pairwise interaction among co-routines. Resume splits its argument into a continuation g and remaining information x. g is the resumption point of the co-routine being resumed. Resume notes the current continuation g0, which is the next resumption point of the resuming co-routine, and and passes g0 together with x to the resumed co-routine.

▷ $\mathsf{Resume} \overset{\mathrm{df}}{\longleftrightarrow} \lambda[\mathsf{g},\mathsf{x}]\mathsf{note}(\mathsf{g0})\mathsf{g}[\mathsf{g0},\mathsf{x}]$

As a particular example, we adapt a segment of network code that uses co-routining (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 co-routine INBYTE responsible for getting the next byte (8-bits) from the 36-bit word stream. Another co-routine 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 8-bit bytes contained in two consecutive 36-bit words. In order to focus on the properties of the co-routine mechanism, we simplify the problem by defining a *Rum* co-routine $\mathsf{ThreeTwoC}(\gamma_{\mathrm{in}})$ which transforms a sequence of strings of length three generated by the co-routine γ_{in} into a sequence of strings of length two with the same characters. If σ is a sequence of strings of length three then $\sigma^{(3,2)}$, the sequence of strings of length two with the same characters as σ, is defined by

$$\begin{aligned}&[\mathsf{StrUn}(\sigma^{(3,2)}(3i)),\ \mathsf{StrUn}(\sigma^{(3,2)}(3i+1)),\ \mathsf{StrUn}(\sigma^{(3,2)}(3i+2))]\\ &\quad = [\mathsf{StrUn}(\sigma(2i)),\ \mathsf{StrUn}(\sigma(2i+1))]\end{aligned}$$

for all numbers i. Using the notions defined above, we can specify the co-routine ThreeTwoC(γ_{in}) as follows.

($\dagger^{(3,2)}$) If σ is a sequences of strings of length three and γ_{in} is a co-routine generating σ then ThreeTwoC(γ_{in}) is a co-routine generating the sequence $\sigma^{(3,2)}$.

To define ThreeTwoC we first define the pfn ThreeTwo that describes the computation to be carried out. ThreeTwo has two arguments, an input co-routine in and an output co-routine out. Initially ThreeTwo reads a string, stores the three characters as [x0, x1, x2], and resumes out, passing the string with elements [x0, x1]. When next resumed, ThreeTwo gets a second string from in, stores the three characters as [x3, x4, x5], and resumes out, passing the string with elements [x2, x3]. When resumed a third time, ThreeTwo simply resumes out, passing the string with elements [x4, x5]. When resumed a fourth time, ThreeTwo repeats its initial computation with the current values of in, out.

$$
\begin{array}{lll}
\triangleright & \mathsf{ThreeTwo} \overset{\mathrm{df}}{\longleftrightarrow} & \mathsf{Rec}(\lambda(\mathsf{Co})\lambda(\mathsf{in})\lambda(\mathsf{out}) \\
& & \quad \mathsf{let}\{[\mathsf{in},\mathsf{w}] \leftarrowtail \mathsf{Resume}[\mathsf{in}]\} \\
& & \quad \mathsf{let}\{[\mathsf{x0},\mathsf{x1},\mathsf{x2}] \leftarrowtail \mathsf{StrUn}(\mathsf{w})\} \\
& & \quad \mathsf{let}\{[\mathsf{out}] \leftarrowtail \mathsf{Resume}[\mathsf{out},\mathsf{StrMk}[\mathsf{x0},\mathsf{x1}]]\} \\
& & \quad \mathsf{let}\{[\mathsf{in},\mathsf{w}] \leftarrowtail \mathsf{Resume}[\mathsf{in}]\} \\
& & \quad \mathsf{let}\{[\mathsf{x3},\mathsf{x4},\mathsf{x5}] \leftarrowtail \mathsf{StrUn}(\mathsf{w})\} \\
& & \quad \mathsf{let}\{[\mathsf{out}] \leftarrowtail \mathsf{Resume}[\mathsf{out},\mathsf{StrMk}[\mathsf{x2},\mathsf{x3})]\} \\
& & \quad \mathsf{let}\{[\mathsf{out}] \leftarrowtail \mathsf{Resume}[\mathsf{out},\mathsf{StrMk}[\mathsf{x4},\mathsf{x5}]]\} \\
& & \qquad \mathsf{Co}(\mathsf{in},\mathsf{out})
\end{array}
$$

ThreeTwoC is a pfn which when applied to an input co-routine γ_{in} generates a co-routine initialized to apply its resumer to ThreeTwo(γ_{in}).

$$\triangleright \quad \mathsf{ThreeTwoC} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{in})\mathsf{ThreeTwo}(\mathsf{in},\mathsf{note}(\mathsf{g})\mathsf{g})$$

In particular

$$\gamma \triangledown \mathsf{ThreeTwoC}'\gamma_{\mathrm{in}} \rightarrowtail \gamma \vartriangle \gamma \circ \mathsf{Appc}(\mathsf{ThreeTwo}(\gamma_{\mathrm{in}}))$$

Since the context in which ThreeTwoC is initialized does not effect the behavior of resulting co-routine, we let ThreeTwoC(γ_{in}) denote Id ∘ Appc(ThreeTwo(γ_{in})) in the following.

Proof of correctness of ThreeTwoC. We outline the proof that ThreeTwoC satisfies ($\dagger^{(3,2)}$) to illustrate basic properties of resumption, and the general idea of such proofs. Let σ be as sequence of strings of length three and let γ_{in} be a co-routine generating σ. Let In be the sequence of continuations corresponding to the generation of σ by γ_{in}. Thus $In(0) = \gamma_{\text{in}}$ and $In(i)'\gamma \rightarrowtail \gamma'[In(i+1), \sigma(i)]$ for any γ and number i. Now we need only define a sequence of continuations Γ such that

$$\Gamma(0) = \mathsf{Id} \circ \mathsf{Appc}(\mathsf{ThreeTwo}(In(0)))$$

$$\Gamma(i)'\gamma \rightarrowtail \gamma'[\Gamma(i+1), \sigma^{(3,2)}(i)]$$

If s is a string we write $s{\downarrow_j}$ for the j-th character of s – $s{\downarrow_j} \underset{\text{df}}{=} \mathsf{StrUn}(s){\downarrow_j}$. Let ch be the sequence of characters from σ. Then $ch(3i+j) = \sigma(i){\downarrow_j}$ for $0 \leq j \leq 2$ and $ch(2i+j) = \sigma^{(3,2)}(i){\downarrow_j}$ for $0 \leq j \leq 1$. We define Γ to be the sequence satisfying

$$\Gamma(3i) = \mathsf{Id} \circ \mathsf{Appc}(\mathsf{ThreeTwo}(In(2i)))$$

$$\Gamma(3i+1) = \mathsf{Id} \circ \mathsf{Appc}(\mathsf{ThreeTwo1}(In(2i+1), ch(6i+2)))$$

$$\Gamma(3i+2) = \mathsf{Id} \circ \mathsf{Appc}(\mathsf{ThreeTwo2}(In(2i+2), ch(6i+4), ch(6i+5)))$$

where

```
ThreeTwo1 = λ(in,x2)λ(out)
                    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]]}
                        ThreeTwo(in, out)
ThreeTwo2 = λ(in,x4, x5)λ(out)
                    let{[out] ↞ Resume[out, StrMk[x4, x5]]}
                        ThreeTwo(in, out)
```

Using the equations relating ch, σ, and $\sigma^{(3,2)}$, and the properties of let and Resume given below it is easy to show that

$$\Gamma(3i)'\gamma \rightarrowtail \gamma'[\Gamma(3i+1), \sigma^{(3,2)}(3i)]$$

$$\Gamma(3i+1)'\gamma \rightarrowtail \gamma'[\Gamma(3i+2), \sigma^{(3,2)}(3i+1)]$$

$$\Gamma(3i+2)'\gamma \rightarrowtail \gamma'[\Gamma(3(i+1)), \sigma^{(3,2)}(3i+2).]$$

Hence, Γ is the required sequence of continuations verifying that $\mathsf{ThreeTwoC}(\gamma_{\text{in}})$ is a co-routine generating $\sigma^{(3,2)}$. $\square_{(\dagger^{(3,2)})}$

■ **Properties of let and Resume.** Unwinding the definitions and applying the step rules we have

(1) $\langle \mathfrak{f}_0 \mid \xi \rangle \hookrightarrow v_0 \rightarrow \gamma \triangledown \langle \mathsf{let}\{s \leftarrow \mathfrak{f}_0\}\mathfrak{f}_1 \mid \xi \rangle \rightarrowtail \gamma \triangledown \langle \mathfrak{f}_1 \mid \xi\{s \leftarrow \mathfrak{f}_0\} \rangle$

(2) $\langle \mathfrak{f}_0 \mid \xi \rangle \hookrightarrow v_0 \rightarrow$

$$\gamma \triangledown \langle \mathsf{let}\{[s_{\mathrm{co}}, s_{\mathrm{v}}] \leftarrow \mathsf{Resume}[s_{\mathrm{co}}, \mathfrak{f}_0]\}\mathfrak{f}_1 \mid \xi \rangle$$

$$\rightarrowtail \gamma \triangledown \xi(s_{\mathrm{co}})\,{}'[\gamma \circ \mathsf{Appc}(\langle \lambda[s_{\mathrm{co}}, s_{\mathrm{v}}]\mathfrak{f}_1 \mid \xi \rangle), v_0]$$

(3) $\langle \mathfrak{f}_0 \mid \xi \rangle \hookrightarrow v_0 \wedge (\exists \gamma_1, v_1)(\forall \gamma_0)(\xi(s_{\mathrm{co}})\,{}'[\gamma_0, v_0] \rightarrowtail \gamma_0{}'[\gamma_1, v_1]) \rightarrow$

$$\gamma \triangledown \langle \mathsf{let}\{[s_{\mathrm{co}}, s_{\mathrm{v}}] \leftarrow \mathsf{Resume}[s_{\mathrm{co}}, \mathfrak{f}_0]\}\mathfrak{f}_1 \mid \xi \rangle$$

$$\rightarrowtail \gamma \triangledown \langle \mathfrak{f}_1 \mid \xi\{s_{\mathrm{co}} \leftarrow \gamma_1, s_{\mathrm{v}} \leftarrow v_1\} \rangle$$

Exercise 1. Streams and co-routines are two mechanisms for generating sequences. These mechanisms are interconvertible. Define pfns Str2Co and Co2Str such that for any infinite sequence σ from the computation domain, if ϑ is a stream generating σ (i.e. $\vartheta^{(i)} = \sigma(i)$) then $\mathsf{Id} \circ \mathsf{Appc}(\mathsf{Str2Co}(\vartheta))$ is a co-routine generating σ and if γ_{co} is a co-routine generating σ then $\mathsf{Co2Str}(\gamma_{\mathrm{co}})$ is a stream generating σ. Prove that

■ **Str2Co is correct.** There is a sequence of continuations Γ such that $\Gamma(0) = \mathsf{Id} \circ \mathsf{Appc}(\mathsf{Str2Co}(\vartheta))$ and for all numbers i, $\Gamma(i)\,{}'[\gamma] \rightarrowtail \gamma{}'[\Gamma(i+1), \sigma(i)]$.

■ **Co2Str is correct.** $\mathsf{Co2Str}(\vartheta)^{(i)} = \sigma(i)$ for all numbers i.

[See §IV.6 for the definition of streams and related concepts.]

Chapter VI. Comparison relations

We now return to the world of tree-structured computation to study a class of relations called *comparison* relations. These allow one to compare one pfn to another or, more generally, to compare dtrees or forms, selectively forgetting some details of the computations described while preserving the applicative structure and the evaluation relation. In particular, comparable dtrees, when defined, have comparable values, and comparable pfns applied to comparable values are comparable dtrees.

There are several reasons for developing a theory of comparison relations. One reason is to satisfy a purely mathematical curiosity as to what meaningful notions of comparison exist, what the structure of this class of relations is, and what 'equations' and 'approximations' hold for particular comparison relations. A paradigm for this work is the elegant equational theory developed for LAMBDA by Scott [1976]. Another reason for studying comparisons is to develop a richer language and more powerful tools for stating and proving properties of pfns, and to build a foundation on which to develop a theory of program specifications and transformations in *Rum*. For more specific motivations, consider the following questions.

- In what useful senses can equations such as $\mathsf{F}(\mathsf{x}) \rightleftharpoons \mathfrak{f}$ characterize a pfn F in the case that F does not appear free in $\mathfrak{f}$? Recall that such equations hold for the algebraic combinator pfns.

- Are there equivalence relations closed under substitution? Note that for $\rightleftharpoons$, substitution is limited to positions that are evaluated or eliminated in the course of a computation, since otherwise substituted forms may return substituted, non-equal values.

- Are there comparisons closed under abstraction? If $\mathfrak{f}_0 \sim \mathfrak{f}_1$ and s is a free-variable then $\lambda(s)\mathfrak{f}_0 \sim \lambda(s)\mathfrak{f}_1$?

- The *Rum* recursion theorem (§IV.4) together with computation induction allows us to prove many induction schemes derived from the least fixed point theorem for the graph model of the lambda calculus, but does not capture the full power of this theorem. Is there a partial ordering on pfns such that Rec computes the least fixed point with respect to this ordering?

- What characterizes the recursion pfn? Are there other recursion pfns?

- Can parameters be moved across the recursion boundary in general? (see §IV.4)
- Intuitively *Rum* pfns are extensional in the sense that they can only be applied, bound in environments or returned as values. Further, we have claimed that pfns are descriptions of partial functions. If pfns describe partial functions, then there must be an equivalence relation $\sim$ on the computation domain satisfying $(\forall v)(\varphi_0{}'v \sim \varphi_1{}'v) \rightarrow \varphi_0 \sim \varphi_1$. As demonstrated by the non-extensional theories of Feferman [1975] (see §I.5), such equivalence relations may not exist in general.
- Note that there are pfns φ that satisfy the self property, $\varphi' v \hookrightarrow \varphi$. One example is the pfn $\mathsf{Self} = \mathsf{Rec}(\mathsf{K})$. Do all pfns that satisfy the self property compute the same function?

The comparison relations on dtrees are defined in §1 and additional conditions characterizing special kinds of comparisons are defined. Each comparison on dtrees induces a natural comparison on values and environments. In general comparison relations are approximation relations – since undefined dtrees may be compared to defined dtrees and hence for comparable pfns, the domain of definition of one pfn may be a proper subset of the domain of definition of the other. An important subclass of comparisons are the invertible relations for which comparable dtrees are equi-defined and comparable pfns have the same domain of definition. Basic results about the structure of the set of comparisons relations are given including results about the preservation of operations on and properties of comparisons by algebraic operations such as intersection and transitive closure.

Comparison relations form a rich hierarchy. At the extremes are equality and the maximum comparison $\sqsubseteq$. Equality in *Rum* is the finest grained comparison, and corresponds to forgetting nothing. $\sqsubseteq$ corresponds to forgetting all the details of computation and is analogous to the partial ordering induced by the graph model on expressions of LAMBDA (Scott [1976]). The intersection of $\sqsubseteq$ with its inverse is $\cong$, the maximum symmetric comparison. In §2 we study the relations $\sqsubseteq$ and $\cong$. The main results are the extensionality of $\sqsubseteq$ and $\cong$, a characterization of recursion pfns, and the $\sqsubseteq$-minimality of the fixed point computed by Rec. Some laws for $\cong$ expressed in the language of forms are given together with an example derivation using these laws. This is a small step towards developing a *Rum*-calculus. In §3 an operation extending a comparison to contain a given pfn relation is defined and conditions are given under which the extension is guaranteed to be a comparison. The main application of this extension operation is to prove that a given pfn relation is contained in the maximum comparison or the maximum equivalence. We use this technique to prove the extensionality and fixed point theorems stated in §2. In §4 an operation extending a comparison to contain a given dtree relation is defined and conditions are given under which the extension is guaranteed to be a comparison. Use of this operation is illustrated by defining a several classes of comparisons including comparisons generated by partial evaluation, let-conversion, and normalizing operations on nested if- and cart- forms.

VI.1. Comparisons and closure conditions

The work on comparisons is carried out in t-$\mathcal{R}um$. Thus $\mathbb{D}t$ means t-$\mathbb{D}t$ and $\mathfrak{d}$ ranges over t-$\mathbb{D}t$, etc. For this chapter "dtree relation" means a binary relation on closure dtrees, and unless otherwise mentioned, dtree will mean closure dtree. ρ and χ range over dtree relations and Γ ranges over *non-empty* sets of dtree relations. $\rho_=$ will be used to denote the equality relation on closure dtrees. We use standard notions and operations for binary relations as summarized in §III.7. For the readers convenience we recall two not so standard operations that we will make frequent use of: transitive union ($\uplus\Gamma$) and inversion closure ($\rho^{\sim}$). Transitive union of a set of relations is the transitive reflexive closure of the union over the set, and inversion closure of a relation is the intersection of the relation with its inverse.

Before proceeding we introduce some additional notation for closure dtrees that will simplify discussion of dtree relations. Full details are given in Appendix A. For non-binding constructs, we use dtree construction notation paralleling the form construction notation. For example, if $\mathfrak{d}_0 = \langle \mathfrak{f}_0 \mid \xi \rangle$ and $\mathfrak{d}_1 = \langle \mathfrak{f}_1 \mid \xi \rangle$ then $\mathsf{app}(\mathfrak{d}_0, \mathfrak{d}_1) = \langle \mathsf{app}(\mathfrak{f}_0, \mathfrak{f}_1) \mid \xi \rangle$. Value dtrees $\langle \mathsf{x} \mid \mathsf{x} \leftarrow v \rangle$ are abbreviated as $\triangleleft v$. To express abstraction and substitution we extend the construction of closure dtrees to include dtree contexts. A dtree context is a closure dtree with some holes – some nodes where there are place holders rather than dtrees. We write $\mathfrak{d}\{\ldots\}$ to indicate a dtree context. The substitution of $\mathfrak{d}_0$ in a context $\mathfrak{d}\{\ldots\}$, (written $\mathfrak{d}\{\mathfrak{d}_0\}$), is the dtree obtained by putting $\mathfrak{d}_0$ in each of the holes. The abstraction of a context $\mathfrak{d}\{\ldots\}$, (written $\lambda\mathfrak{d}\{\ldots\}$), is the dtree obtained by replacing the holes with binding pointers to the outer λ. An n-ary dtree context is a dtree context where the holes have been partitioned into n classes - say by labeling them with numbers from 1 to n. Substitution and abstraction generalize in the obvious manner to n-ary dtree contexts, with abstraction referring to the first class of holes, the second class becoming the first remaining class, etc. One can think of dtree contexts as being given by a form, an environment and a list of variables to be held free. Only free symbols in the form which are not in the free list will become value nodes. The symbols held free can be thought of as marking the holes and partitioning them into classes, one for each symbol with the ordering given by the list.

VI.1.1. Dtree comparisons

The basic comparisons are dtree comparisons. Everything else is derived naturally from this class of relations. We begin by defining the extension of a dtree relation to values and environments and the evaluation closure operations on dtree relations. These are useful operations for expressing properties of dtree relations. We then define the general notions of comparison and invertible comparison.

▹ **The extension of a dtree relation to values and environments.** Since we view pfns as λ-dtrees, each dtree relation determines a natural relation on pfns, namely the restriction to pfns. We write $\rho[\mathbb{P}]$ for the restriction of ρ to pfns.

$$\boldsymbol{\partial}_0 \; \rho[\mathbb{P}] \; \boldsymbol{\partial}_1 \underset{\mathbf{df}}{\equiv} \boldsymbol{\partial}_0 \in \mathbb{P} \wedge \boldsymbol{\partial}_1 \in \mathbb{P} \wedge \boldsymbol{\partial}_0 \, \rho \, \boldsymbol{\partial}_1$$

Since we are interested in the structure of computation and wish to maintain uniformity over the given data structure, we require that value comparisons restrict to equality on data and data operations and that sequences and environments are compared component wise. Thus, a dtree relation ρ has a unique extension to values and environments, which we also denote by ρ.

$$u \; \rho \; v \underset{\mathbf{df}}{\equiv} |u| = |v| \wedge (\forall i < |u|)(u{\downarrow}_i = v{\downarrow}_i \vee u{\downarrow}_i \, \rho[\mathbb{P}] \; v{\downarrow}_i)$$

$$\xi_0 \; \rho \; \xi_1 \underset{\mathbf{df}}{\equiv} (\forall s)(\xi_0(s) \; \rho \; \xi_1(s))$$

▹ **Evaluation closure operations.** Evaluation closure of a dtree relation relates two dtrees when they are both defined and have related values. There are two possibilities for the case that the first dtree is undefined. For approximation relations the evaluation closure relates an undefined dtree to any other. For invertible relations we require that the second dtree be undefined also. For a dtree relation ρ, the evaluation closure of ρ, (written $\overline{\rho}$), and the invertible evaluation closure of ρ, (written $\overline{\rho}^{i}$), are defined as follows.

$$\boldsymbol{\partial}_0 \, \overline{\rho} \, \boldsymbol{\partial}_1 \underset{\mathbf{df}}{\equiv} \Downarrow\boldsymbol{\partial}_0 \rightarrow (\exists v_0, v_1)(v_0 \; \rho \; v_1 \wedge \boldsymbol{\partial}_0 \hookrightarrow v_0 \wedge \boldsymbol{\partial}_1 \hookrightarrow v_1)$$

$$\boldsymbol{\partial}_0 \, \overline{\rho}^{i} \, \boldsymbol{\partial}_1 \underset{\mathbf{df}}{\equiv} \Downarrow\boldsymbol{\partial}_0 \leftrightarrow \Downarrow\boldsymbol{\partial}_1 \wedge \boldsymbol{\partial}_0 \, \overline{\rho} \, \boldsymbol{\partial}_1$$

▹ **Dtree comparison relations.** A dtree comparison is a reflexive transitive dtree relation that is a subrelation of its evaluation closure ($C^{e}(\rho)$) and closed under application ($\mathcal{C}^a(\rho)$). $\mathcal{C}$ is the set of dtree comparisons.

$$\rho \in \mathcal{C} \underset{\mathrm{df}}{\equiv} \rho = \rho^* \wedge \mathcal{C}^e(\rho) \wedge \mathcal{C}^a(\rho)$$

where

$$\mathcal{C}^e(\rho) \underset{\mathrm{df}}{\equiv} \rho \subset \overline{\rho}$$

$$\mathcal{C}^a(\rho) \underset{\mathrm{df}}{\equiv} (\forall \varphi_0, \varphi_1, v_0, v_1)(\varphi_0 \, \rho \, \varphi_1 \wedge v_0 \, \rho \, v_1 \rightarrow (\varphi_0 \,' v_0) \, \rho \, (\varphi_1 \,' v_1))$$

■ **Substitution of values.** The point of requiring closure under application in the definition of comparison relation is to insure that a comparison is closed under substitutions of comparable values. For any n-ary dtree context $\mathfrak{d}\{\ldots\}$

$$\rho \in \mathcal{C} \wedge u_{0,1} \, \rho \, u_{1,1} \wedge \ldots \wedge u_{0,\mathrm{n}} \, \rho \, u_{1,\mathrm{n}} \rightarrow \mathfrak{d}\{\triangleleft u_{0,1} \ldots \triangleleft u_{0,\mathrm{n}}\} \, \rho \, \mathfrak{d}\{\triangleleft u_{1,1} \ldots \triangleleft u_{1,\mathrm{n}}\}$$

Expressed in terms of forms closed in related environments this becomes.

$$\rho \in \mathcal{C} \wedge \xi_0 \, \rho \, \xi_1 \rightarrow \langle \mathfrak{f} \mid \xi_0 \rangle \, \rho \, \langle \mathfrak{f} \mid \xi_1 \rangle$$

▷ **Invertible comparisons.** An *invertible* comparison is a comparison whose inverse is also a comparison. $\mathcal{C}_{\mathrm{i}}$ denotes the set of invertible comparisons.

$$\rho \in \mathcal{C}_{\mathrm{i}} \underset{\mathrm{df}}{\equiv} \rho \in \mathcal{C} \wedge \rho^{-} \in \mathcal{C}$$

Examples. Two familiar dtree relations, dtree equality ($\rho_=$) and ctree equivalence ($\rightleftharpoons$) are easily seen to be comparisons. In fact both are symmetric and hence invertible. Since comparisons are reflexive, dtree equality is a subrelation of every comparison, i.e. it is the minimal comparison. Note also that $\rightleftharpoons = \overline{\rho_=}^{\,i}$.

VI.1.2. About evaluation closure

The following are simple but useful consequences of the definitions of the extension of dtree relations to values, the evaluation closure operations, and comparisons. We will refer to these facts as "basic properties of evaluation closure".

■ The evaluation closure operations are monotonic and fix the pfn component of a relation.

$$\rho_0 \subset \rho_1 \rightarrow \overline{\rho_0} \subset \overline{\rho_1} \wedge \overline{\rho_0}^{\,i} \subset \overline{\rho_1}^{\,i}$$

$$\overline{\rho}[\mathbb{P}] = \overline{\rho}^{\,i}[\mathbb{P}] = \rho[\mathbb{P}]$$

■ The [invertible] evaluation closure of a relation is determined by its restriction to pfns and hence the evaluation closure operations are idempotent.

$$\overline{\rho} = \overline{\rho[\mathbb{P}]} \quad \text{and} \quad \overline{\rho}^{i} = \overline{\rho[\mathbb{P}]}^{i}$$

$$\overline{\overline{\rho}} = \overline{\rho} \quad \text{and} \quad \overline{\overline{\rho}^{i}}^{i} = \overline{\rho}^{i}$$

■ The invertible evaluation closure of a relation is the intersection of the evaluation closure of the relation with the inverse of the evaluation closure of its inverse.

$$\overline{\rho}^{i} = \overline{\rho} \cap \left(\overline{\rho^{-}}\right)^{-}$$

■ The [invertible] evaluation closure of a[n invertible] comparison ρ is a[n invertible] comparison. It is the maximum [invertible] comparison whose restriction to pfns is the restriction of ρ to pfns.

$$\rho \in \mathcal{C} \rightarrow \overline{\rho} \in \mathcal{C} \quad \text{and} \quad \rho \in \mathcal{C}_{\mathrm{i}} \rightarrow \overline{\rho}^{i} \in \mathcal{C}_{\mathrm{i}}$$

$$\rho_0 \in \mathcal{C} \wedge \rho_1 \in \mathcal{C} \wedge \rho_0[\mathbb{P}] = \rho_1[\mathbb{P}] \rightarrow \rho_1 \subset \overline{\rho_0}$$

$$\rho_0 \in \mathcal{C}_{\mathrm{i}} \wedge \rho_1 \in \mathcal{C}_{\mathrm{i}} \wedge \rho_0[\mathbb{P}] = \rho_1[\mathbb{P}] \rightarrow \rho_1 \subset \overline{\rho_0}^{i}$$

VI.1.3. Closure conditions for comparisons

We now define three closure conditions for dtree relations: evaluation-closed, extensionally-closed, and substitution-closed. These closure conditions serve to identify additional interesting sets of dtree comparisons. We give four simple comparisons to demonstrate that for each closure condition there are comparisons which satisfy it and comparisons which fail to satisfy it. We show that the evaluation closure operation on comparisons is (as the name implies) a closure operation for the evaluation closure condition – the evaluation closure of a comparison is the least evaluation-closed comparison containing the given comparison. We will see later that there are also closure operations for the dtree substitution and extensional closure conditions. Finally we show that the evaluation closure together with extensional closure imply substitution closure.

▷ **Evaluation-closed.** A dtree relation ρ is [invertible] evaluation-closed if it contains its [invertible] evaluation closure.

$$Cl_{ev}(\rho) \underset{\mathrm{df}}{\equiv} \overline{\rho} \subset \rho$$

$$Cl_{iev}(\rho) \underset{\mathrm{df}}{\equiv} \overline{\rho}^{i} \subset \rho$$

▷ **Extensionally-closed.** A dtree relation is extensionally-closed if two pfns that give related dtrees when applied to any value are related

$$Cl_{ext}(\rho) \underset{\mathrm{df}}{\equiv} (\forall \varphi_0, \varphi_1)((\forall v)(\varphi_0{}' v \;\rho\; \varphi_1{}' v) \rightarrow \varphi_0 \;\rho\; \varphi_1)$$

▷ **Substitution-closed.** A dtree relation is substitution-closed if substitution of related dtrees into a dtree context gives related dtrees.

$$Cl_{dsubst}(\rho) \underset{\mathrm{df}}{\equiv} (\forall \mathfrak{d}\{\ldots\}, \mathfrak{d}_0, \mathfrak{d}_1)(\mathfrak{d}_0 \;\rho\; \mathfrak{d}_1 \rightarrow \mathfrak{d}\{\mathfrak{d}_0\} \;\rho\; \mathfrak{d}\{\mathfrak{d}_1\})$$

■ **Examples of closure and non-closure.** Let $\rho_0 = \overline{\rho_=}$, $\rho_1 = \overline{\rho_=}^{\,i}$ and $\rho_2 = \rho_= \cup \{(\langle \mathsf{x} \mid \mathsf{x} \leftarrow v\rangle, \langle \mathsf{I}(\mathsf{x}) \mid \mathsf{x} \leftarrow v\rangle)\}$. Then $\rho_=$, ρ_0, ρ_1, and ρ_2 are comparisons and we have

$$\begin{array}{llll}
\neg Cl_{ev}(\rho_=) & Cl_{ev}(\rho_0) & \neg Cl_{ev}(\rho_1) & \neg Cl_{ev}(\rho_2) \\
\neg Cl_{iev}(\rho_=) & \neg Cl_{iev}(\rho_0) & Cl_{iev}(\rho_1) & \neg Cl_{iev}(\rho_2) \\
Cl_{dsubst}(\rho_=) & \neg Cl_{dsubst}(\rho_0) & \neg Cl_{dsubst}(\rho_1) & \neg Cl_{dsubst}(\rho_2) \\
Cl_{ext}(\rho_=) & \neg Cl_{ext}(\rho_0) & \neg Cl_{ext}(\rho_1) & \neg Cl_{ext}(\rho_2)
\end{array}$$

■ **The evaluation closure is evaluation-closed.** From the basic evaluation closure properties we see that if ρ is a comparison then the evaluation closure is the least evaluation-closed comparison containing ρ. Similarly for invertible comparisons and invertible evaluation closure.

$$\rho \in \mathcal{C} \rightarrow \overline{\rho} \in \mathcal{C} \wedge Cl_{ev}(\overline{\rho}) \quad \text{and} \quad \rho \subset \rho_0 \in \mathcal{C} \wedge Cl_{ev}(\overline{\rho_0}) \rightarrow \overline{\rho} \subset \rho_0$$

$$\rho \in \mathcal{C}_{\mathrm{i}} \rightarrow \overline{\rho}^{\,i} \in \mathcal{C}_{\mathrm{i}} \wedge Cl_{iev}(\overline{\rho}^{\,i}) \quad \text{and} \quad \rho \subset \rho_0 \in \mathcal{C}_{\mathrm{i}} \wedge Cl_{iev}(\overline{\rho_0}^{\,i}) \rightarrow \overline{\rho}^{\,i} \subset \rho_0$$

■ **Non-independence for closure conditions.** A dtree comparison that is evaluation-closed and extensionally-closed is also substitution-closed. Similarly for the invertible case.

$$\rho \in \mathcal{C} \wedge Cl_{ev}(\rho) \wedge Cl_{ext}(\rho) \rightarrow Cl_{dsubst}(\rho)$$

$$\rho \in \mathcal{C}_{\mathrm{i}} \wedge Cl_{iev}(\rho) \wedge Cl_{ext}(\rho) \rightarrow Cl_{dsubst}(\rho)$$

Proof: by induction on the structure of dtree contexts, using the definitions of evaluation, comparison and of the closure conditions.

VI.1.4. Algebraic operations on comparison relations

We will make frequent use of algebraic operations on relations and sets of relations. Of particular interest are transitive union, intersection over non-empty sets, inversion, and inversion closure. Note that $\rho^* = \uplus\{\rho\}$ so the transitive reflexive closure operation is also covered. Figure 18 contains a summary of operations on dtree relations that are preserved by (commute with) the algebraic operations and a summary of properties of dtree relations that are preserved by these operations. For the laws concerning transitive union to hold we must restrict the domain to non-empty sets of reflexive relations ρ satisfying C^e. This is not a serious restriction as it includes the intended domains of application. The commuting and preservation laws are mainly a direct consequence of the definitions involved. We will prove some of the laws for transitive union to illustrate how the proofs go in general.

Operation	Preserved by[†]	Remarks
$\rho[\mathbb{P}]$	$\uplus, \cap, ^-, ^\sim$	
$\overline{\rho}$	$\uplus, \cap$	$(\overline{\rho})^\sim = (\overrightarrow{\rho}^{i})^\sim$
$\overrightarrow{\rho}^{i}$	$\uplus, \cap, ^-, ^\sim$	

Property	Preserved by	Remarks
$\rho = \rho^*$	$\uplus, \cap, ^-, ^\sim$	C^e not needed
$C^e(\rho)$	$\uplus, \cap, ^\sim$	$\rho \subset \overline{\rho} \rightarrow \rho^\sim \subset \overrightarrow{\rho^\sim}^{i}$
$\rho \subset \overrightarrow{\rho}^{i}$	$\uplus, \cap, ^-, ^\sim$	
$C^a(\rho)$	$\uplus, \cap, ^-, ^\sim$	
$\rho \in \mathcal{C}$	$\uplus, \cap, ^\sim$	$\rho \in \mathcal{C} \rightarrow \rho^\sim \in \mathcal{C}_i$
$\rho \in \mathcal{C}_i$	$\uplus, \cap, ^-, ^\sim$	
$Cl_{ev}(\rho)$	$\uplus, \cap, ^\sim$	$Cl_{ev}(\rho) \rightarrow Cl_{iev}(\rho^\sim)$
$Cl_{iev}(\rho)$	$\uplus, \cap, ^-, ^\sim$	
$Cl_{ext}(\rho)$	$\cap, ^-, ^\sim$	
$Cl_{dsubst}(\rho)$	$\uplus, \cap, ^-, ^\sim$	C^e not needed

[†] Γ non-empty; $\uplus$ restricted to Γ such that $\rho \in \Gamma \rightarrow \rho_= \subset \rho \wedge C^e(\rho)$

Figure 18. Preservation of comparison operations and properties

First, to make the content of this summary clear we give the reading of the entries for evaluation closure and for comparisons. The entry for evaluation closure is an abbreviation for the following three statements.

$$(\forall\rho\in\Gamma)(\rho_{=}\subset\rho\wedge C^{e}(\rho)\rightarrow\uplus\overline{\Gamma}=\overline{\uplus\Gamma})$$
$$\cap\overline{\Gamma}=\overline{\cap\Gamma}$$
$$(\overline{\rho})^{\sim}=(\overline{\rho}^{i})^{\sim}$$

The entry for comparisons is an abbreviation for the following three statements.

$$\Gamma\subset\mathcal{C}\rightarrow\uplus\Gamma\in\mathcal{C}$$
$$\Gamma\subset\mathcal{C}\rightarrow\cap\Gamma\in\mathcal{C}$$
$$\rho\in\mathcal{C}\rightarrow\rho^{\sim}\in\mathcal{C}_{\mathrm{i}}$$

Proofs for transitive union.

The proofs generally have the form of a chain of equivalences or implications. The justification for a step is given in small print to the side or below the deduced formula and is preceded by the sign %.

We first prove the commuting laws for transitive union. The key is commuting with the restriction to pfns. The requirement that Γ satisfy $\rho\in\Gamma\rightarrow C^{e}(\rho)$ is essential here. Commuting with the restriction to pfns and requirement that Γ satisfy $\rho\in\Gamma\rightarrow\rho_{=}\subset\rho$ are needed to prove commuting with the extension to values. In turn commuting with the restriction to pfns and the extension to values are essential in proving the remaining preservation laws.

■ **Commuting laws for transitive union.** Let Γ be a non-empty set of reflexive relations that satisfy C^{e}. Then

(i) $(\uplus\Gamma)[\mathbb{P}]=\uplus(\Gamma[\mathbb{P}])$

(ii) $v_0\uplus\Gamma\,v_1\leftrightarrow(\exists k)(\exists\{\rho_1\ldots\rho_{\mathrm{k}}\}\subset\Gamma)(\exists u_0\ldots u_{\mathrm{k}})(v_0=u_0\,\rho_1\ldots\rho_{\mathrm{k}}\,u_{\mathrm{k}}=v_1)$

(iii) $\overline{\uplus\Gamma}=\uplus\overline{\Gamma}$

(iv) $\overline{\uplus\Gamma}^{i}=\uplus\overline{\Gamma}^{i}$

Proof(i):

$\partial_a\,(\uplus\Gamma)\;[\mathbb{P}]\;\partial_b$

$\leftrightarrow\;\partial_a \in \mathbb{P} \wedge \partial_b \in \mathbb{P} \wedge$

$(\exists k)(\exists\{\rho_0\ldots\rho_k\} \subset \Gamma)(\exists\partial_0 = \partial_a\ldots\partial_{k+1} = \partial_b)(\forall i \le k)(\partial_i\;\rho_i\;\partial_{i+1})$

% definition of transitive union and restriction to pfns

$\leftrightarrow\;(\exists k)(\exists\{\rho_0\ldots\rho_k\} \subset \Gamma)(\exists\partial_0 = \partial_a\ldots\partial_{k+1} = \partial_b)(\forall i \le k)(\partial_i\;\rho_i\,[\mathbb{P}]\;\partial_{i+1})$

% since ρ_0 and ρ_1 satisfy C^e we have $\varphi_0\;\rho_0\;\partial\;\rho_1\;\varphi_1 \rightarrow (\exists\varphi)(\varphi_0\;\rho_0\;\varphi\;\rho_1\;\varphi_1)$

$\leftrightarrow\;\partial_a\;\uplus(\Gamma\,[\mathbb{P}])\;\partial_b$ % definition of transitive union

$\square_i$

Proof(ii):

$v_0\;\uplus\Gamma\;v_1$

$\leftrightarrow\;|v_0| = |v_1| \wedge (\forall i < |v_0|)(v_0{\downarrow}_i = v_1{\downarrow}_i \vee v_0{\downarrow}_i\,(\uplus\Gamma)\,[\mathbb{P}]\;v_1{\downarrow}_i)$

% definition of canonical extension to values

$\leftrightarrow\;|v_0| = |v_1| \wedge (\forall i < |v_0|)(v_0{\downarrow}_i = v_1{\downarrow}_i \vee v_0{\downarrow}_i\;\uplus(\Gamma\,[\mathbb{P}])\;v_1{\downarrow}_i)$

% by (i)

$\leftrightarrow\;(\exists k)(\exists\{\rho_1\ldots\rho_k\} \subset \Gamma)(\exists u_0\ldots u_k)(v_0 = u_0\;\rho_1\;\ldots\;\rho_k\;u_k = v_1)$

% reflexivity

$\square_{ii}$

Proof(iii):

$\partial_a\;\uplus\;\overline{\Gamma}\;\partial_b$

$\leftrightarrow\;(\exists k)(\exists\{\rho_0\ldots\rho_k\} \subset \Gamma)(\exists\partial_0 = \partial_a\ldots\partial_{k+1} = \partial_b)(\forall i \le k)(\partial_i\;\overline{\rho_i}\;\partial_{i+1})$

% definition of transitive union

$\leftrightarrow\;\partial_a \hookrightarrow v_a \rightarrow (\exists k)(\exists\{\rho_0\ldots\rho_k\} \subset \Gamma)(\exists v_0 = v_a\ldots v_{k+1})$

$(\partial_b \hookrightarrow v_{k+1} \wedge (\forall i \le k)(v_i\;\rho_i\;v_{i+1}))$

% definition of evaluation closure

$\leftrightarrow\;\partial_a \hookrightarrow v_a \rightarrow (\exists v_b)(v_a\;\uplus\Gamma\;v_b \wedge \partial_b \hookrightarrow v_b)$ % using (ii)

$\leftrightarrow\;\partial_a\;\overline{\uplus\Gamma}\;\partial_b$ % definition of evaluation closure

$\square_{iii}$

Proof(iv):

$$\partial_a \uplus \overline{\Gamma}^{i}\, \partial_b$$

$\leftrightarrow$ $(\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_a \ldots \partial_{k+1} = \partial_b)(\forall i \leq k)(\partial_i\, \overline{\rho_i}^{i}\, \partial_{i+1})$

% definition of transitive union

$\leftrightarrow$ $(\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_a \ldots \partial_{k+1} = \partial_b)$
$((\Downarrow\partial_a \leftrightarrow \Downarrow\partial_b) \wedge (\forall i \leq k)(\partial_i\, \overline{\rho_i}\, \partial_{i+1}))$

% definition of invertible evaluation closure

$\leftrightarrow$ $(\Downarrow\partial_a \leftrightarrow \Downarrow\partial_b) \wedge \partial_a \uplus \overline{\Gamma}\, \partial_b$ % definition of transitive union

$\leftrightarrow$ $(\Downarrow\partial_a \leftrightarrow \Downarrow\partial_b) \wedge \partial_a\, \overline{\uplus\Gamma}\, \partial_b$ % by (iii)

$\leftrightarrow$ $\partial_a\, \overline{\uplus\Gamma}^{i}\, \partial_b$ % definition of invertible evaluation closure

$\Box_{iv}$

Now we are ready to prove the preservation laws. We will consider only those laws leading up to the preservation of comparisons and evaluation closure. The corresponding results for the invertible case are proved by replacing evaluation closure by invertible evaluation closure. Preservation of substitution-closure by transitive union (of any non-empty set of relations) is a trivial consequence of the definitions. Here we will be briefer, omitting justifications such as " by definition of transitive union".

■ **Preservation laws for transitive union.** Let Γ be a non-empty set of reflexive relations satisfying C^e. Then

(v) $\uplus\Gamma \subset \overline{\uplus\Gamma}$

(vi) $C^a(\Gamma) \rightarrow C^a(\uplus\Gamma)$

(vii) $\Gamma \subset C \rightarrow \uplus\Gamma \in C$

(viii) $Cl_{ev}(\Gamma) \rightarrow Cl_{ev}(\uplus\Gamma)$

Proof(v):

$\partial_a \uplus\Gamma\, \partial_b$

$\rightarrow (\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_b \ldots \partial_{k+1} = \partial_b)(\forall i \leq k)(\partial_i\ \rho_i\ \partial_{i+1})$

$\rightarrow (\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_a \ldots \partial_{k+1} = \partial_b)(\forall i \leq k)(\partial_i\ \overline{\rho_i}\ \partial_{i+1})$

% since $\rho \in \Gamma \rightarrow \rho \subset \overline{\rho}$.

$\rightarrow \partial_a \uplus \overline{\Gamma}\, \partial_b$

$\rightarrow \partial_a\, \overline{\uplus\Gamma}\, \partial_b$ % by (iii)

$\square_v$

Proof(vi): Assume $C^a(\Gamma)$

$\varphi_a \uplus\Gamma\, \varphi_b \wedge u_a \uplus\Gamma\, u_b$

$\rightarrow (\exists k)(\exists\{\rho_0 \ldots \rho_{k+m+1}\} \subset \Gamma)$

$(\exists \varphi_0 = \varphi_a \ldots \varphi_{k+1} = \varphi_b u_0 = u_a \ldots u_{m+1} = u_b)$

$((\forall i \leq k)(\varphi_i{}' u_0\ \rho_i\ \varphi_{i+1}{}' u_0) \wedge$

$(\forall i \leq m)(\varphi_{k+1}{}' u_i\ \rho_{i+k+1}\ \varphi_{k+1}{}' u_{i+1}))$

% by (i),(ii), and $C^a(\Gamma)$

$\rightarrow \varphi_a{}' u_a \uplus\Gamma\, \varphi_b{}' u_b$

$\square_{vi}$

Proof(vii): by (v) and (vi) since the transitive union of a set is by definition reflexive and transitive.

Proof(viii): Assume $Cl_{ev}(\Gamma)$

$\partial_a\, \overline{\uplus\Gamma}\, \partial_b$

$\rightarrow \partial_a \uplus \overline{\Gamma}\, \partial_b$ % by (iii)

$\rightarrow (\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_a \ldots \partial_{k+1} = \partial_b)(\forall i \leq k)(\partial_i\ \overline{\rho_i}\ \partial_{i+1})$

$\rightarrow (\exists k)(\exists\{\rho_0 \ldots \rho_k\} \subset \Gamma)(\exists \partial_0 = \partial_a \ldots \partial_{k+1} = \partial_b)(\forall i \leq k)(\partial_i\ \rho_i\ \partial_{i+1})$

% evaluation closure of Γ

$\rightarrow \partial_a \uplus\Gamma\, \partial_b$

$\square_{viii}$

VI.2. The maximum comparisons

An important consequence of the closure of comparison relations under transitive union is the existence of maximum comparisons. We define three relations: $\sqsubseteq$, the maximum element of $\mathcal{C}$, $\sqsupseteq$, the inverse of $\sqsubseteq$, and $\cong$, the intersection of $\sqsubseteq$ and its inverse.

▷ **The maximum comparisons.**

$$\sqsubseteq \underset{\text{df}}{=} \uplus\mathcal{C}$$

$$\sqsupseteq \underset{\text{df}}{=} \sqsubseteq^{-}$$

$$\cong \underset{\text{df}}{=} \sqsubseteq \cap \sqsupseteq$$

The following are a few facts about the maximum comparisons.

▪ **Maximality.** By the preservation laws for algebraic operations (Figure18) $\sqsubseteq$ is the maximum comparison and $\cong$ is the maximum invertible comparison.

$$\sqsubseteq \in \mathcal{C} \quad \text{and} \quad \rho \in \mathcal{C} \rightarrow \rho \subset \sqsubseteq$$

$$\cong \in \mathcal{C}_{\mathrm{i}} \quad \text{and} \quad \cong = \cong^{-} \quad \text{and} \quad \rho \in \mathcal{C}_{\mathrm{i}} \rightarrow \rho \subset \cong$$

▪ **Evaluation closure.** By basic properties of evaluation closure $\sqsubseteq$ is evaluation-closed and $\cong$ is invertible evaluation-closed.

$$\overline{\sqsubseteq} = \sqsubseteq \quad \text{and} \quad \overline{\cong}^{i} = \cong$$

The next two facts illustrate that the maximum comparisons are non-trivial in the sense that they do not relate too many pfns.

▪ **Non-triviality 1.** Let Bot be $\mathsf{Rec}(\lambda(\mathsf{f},\mathsf{x})\mathsf{f}(\mathsf{x}))$ a pfn that is everywhere undefined. Then Bot $\sqsubseteq$-approximates every pfn, and no pfn with non-empty domain $\sqsubseteq$-approximates Bot.

$$\mathsf{Bot} \sqsubseteq \varphi \quad \text{and} \quad (\exists v)(\Downarrow \varphi' v) \rightarrow \varphi \not\sqsubseteq \mathsf{Bot}$$

Hence Bot is not $\cong$-equivalent to any pfn with a non-empty domain of definition.

▪ **Non-triviality 2.** Define the numeral sequence $Nu(n)$ by $Nu(0) = \square$ and $Nu(n+1) = \mathsf{K}(Nu(n))$. Then no element of the sequence Nu is $\cong$-equivalent to any other element.

$$m < n \rightarrow Nu(m) \not\cong Nu(n)$$

We prove this to give an idea of one sort of argument used to prove non-equivalence. The proof is by induction on m. For $n > 0$, $Nu(n) \not\cong Nu(0)$ since $Nu(n)$ is a pfn and, by definition of the extension of dtree relation to values, no pfn is $\cong$-equivalent to the empty sequence. For $0 < m < n$ assume $N(m-1) \not\cong Nu(n-1)$. If $Nu(m) \cong Nu(n)$ then $Nu(m)'\square \cong Nu(n)'\square$ by definition of comparison. But $Nu(m)'\square \hookrightarrow Nu(m-1)$ and $Nu(n)'\square \hookrightarrow Nu(n-1)$ so by definition of comparison $N(m-1) \cong Nu(n-1)$ □.

■ **Extensionality Theorem** $\sqsubseteq$ and $\cong$ are extensionally closed.

$$(\text{ext.}\sqsubseteq) \quad (\forall v)(\varphi_0{}'v \sqsubseteq \varphi_1{}'v) \leftrightarrow \varphi_0 \sqsubseteq \varphi_1)$$

$$(\text{ext.}\cong) \quad (\forall v)(\varphi_0{}'v \cong \varphi_1{}'v) \leftrightarrow \varphi_0 \cong \varphi_1)$$

(ext.$\sqsubseteq$) is proved in §3 after some general tools are developed. (ext.$\cong$) is an immediate consequence of (ext.$\sqsubseteq$), the preservation laws, and the definition of $\cong$ as the intersection of $\sqsubseteq$ and its inverse.

■ **Corollary.** By the non-independence lemma $\sqsubseteq$ and $\cong$ are substitution-closed.

$$Cl_{dsubst}(\sqsubseteq) \quad \text{and} \quad Cl_{dsubst}(\cong)$$

VI.2.1. Pfns as partial functions

Since $\cong$ is extensional, pfns can be interpreted as partial functions on the computation domain modulo $\cong$. For any $\mathcal{R}um$ domain A and any element a of A, $\lfloor \mathsf{A} \rfloor_{\cong}$ denotes the set of equivalence classes of A modulo $\cong$ and $\lfloor a \rfloor_{\cong}$ denotes the equivalence class containing a. For each pfn φ we define a partial function $\{\varphi\}_{\cong}$ on $\lfloor \mathbf{V}^* \rfloor_{\cong}$.[1]

$$(\text{fun}) \qquad \{\varphi\}_{\cong}(\lfloor v \rfloor_{\cong}) \rightsquigarrow \lfloor u \rfloor_{\cong} \underset{\text{df}}{\equiv} (\exists u_0 \cong u)(\varphi' v \hookrightarrow u_0)$$

Using properties of symmetric comparisons it is easy to see that (fun) does indeed determine a partial function on $\lfloor \mathbf{V}^* \rfloor_{\cong}$. We call $\{\varphi\}_{\cong}$ the function computed by φ. Extensionality of $\cong$ implies that two pfns compute the same function iff they are $\cong$-equivalent. Thus $\lfloor \mathbb{P} \rfloor_{\cong}$ is a subset of $\lfloor \mathbf{V}^* \rfloor_{\cong}$ which is isomorphic to a partial function space on $\lfloor \mathbf{V}^* \rfloor_{\cong}$. Furthermore $\lfloor \mathbf{V}^* \rfloor_{\cong}$ contains an isomorphic copy of the given data structure $\mathfrak{D}$, and $\lfloor \mathbb{P} \rfloor_{\cong}$ has all the closure properties needed for a theory of computable partial functions over $\mathfrak{D}$, including computable functions of higher types.

[1] Here we are again using braces according to the notation of Kleene [1952]. Context will distinguish between the use of braces in this sense and our normal use as readability braces around function expressions.

This interpretation of pfns justifies the description of pfns as partial functions that contain information about how they are to be computed. It also allows us to use the usual tools for reasoning about partial functions (i.e. about functional binary relations) when working modulo $\cong$. Two simple examples are the laws for identity and composition, and the uniqueness of the "bottom" pfn.

■ **Laws for identity and composition.** The laws for I and $\circ$ given in §IV.4 show that I computes the identity function and that $\circ$ computes the composition functional.[2] It follows that I and $\circ$ satisfy the laws for a monoid. Namely, I is the right and left identity for composition and composition is associative.

$$\mathsf{I} \circ \vartheta \cong \vartheta \cong \vartheta \circ \mathsf{I}$$
$$(\vartheta_0 \circ \vartheta_1) \circ \vartheta_2 \cong \vartheta_0 \circ (\vartheta_1 \circ \vartheta_2)$$

■ **Uniqueness of the bottom pfn.** There is a unique partial function $\perp$ (bottom) on $\mathbf{V}^*$ that is everywhere undefined. The pfn Bot $(\mathsf{Rec}(\lambda(\mathsf{f},\mathsf{x})\mathsf{f}(\mathsf{x})))$ is everywhere undefined and hence computes $\perp$. Thus, a pfn computes $\perp$ iff it is $\cong$-equivalent to Bot.

$$\{\varphi\}_{\cong} = \perp \;\leftrightarrow\; \varphi \cong \mathsf{Bot}$$

VI.2.2. The recursion theorem revisited

Using the maximum comparison we give a strengthened version of the recursion theorem and two consequences – uniqueness of the recursion pfn and a general result for recursion with parameters.

▷ **Recursion pfns.** We say that a pfn φ_{rec} is a *recursion pfn* if it satisfies the following computational laws.

(rec.pfnl) $\varphi_{\mathrm{rec}} \in \mathit{Pfnl}$

(rec.subc) $(\forall \vartheta \in \mathit{Pfnl})(\vartheta\,'(\varphi_{\mathrm{rec}}(\vartheta)) \prec \varphi_{\mathrm{rec}}(\vartheta)\,'\, v)$

(rec.redt) $(\forall \vartheta \in \mathit{Pfnl})(\varphi_{\mathrm{rec}}(\vartheta)\,'\, v \gg \vartheta(\varphi_{\mathrm{rec}}(\vartheta))\,'\, v)$

Note that these are just the laws given for Rec in §IV.4, hence Rec is a recursion pfn.

■ **Improved Recursion theorem.** If φ_{rec} is a recursion pfn then for any pfnl ϑ, φ_{rec} computes the least fixed point of ϑ.

(rec.fix) $\varphi_{\mathrm{rec}}(\vartheta) \cong \vartheta(\varphi_{\mathrm{rec}}(\vartheta))$

(rec.min) $\vartheta(\varphi) \sqsubseteq \varphi \rightarrow \varphi_{\mathrm{rec}}(\vartheta) \sqsubseteq \varphi$

[2] Recall, I is defined as $\lambda(\mathsf{x})\mathsf{x}$ and $\circ$ is infix notation for B which defined as $\lambda(\mathsf{f},\mathsf{g},\mathsf{x})\mathsf{f}(\mathsf{g}(\mathsf{x}))$.

By the definition of recursion pfn $\varphi_{\mathrm{rec}}(\vartheta) \rightleftharpoons \vartheta(\varphi_{\mathrm{rec}}(\vartheta))$ and (rec.fix) follows by maximality of $\cong$. (rec.min) is a consequence of maximality and will be proved in §3. To see that in general the least fixed point of a pfnl is not the only fixed point consider the pfn $\lambda(\mathsf{f},\mathsf{x})\mathsf{f}(\mathsf{x})$. Every pfn satisfies $\lambda(\mathsf{f},\mathsf{x})\mathsf{f}(\mathsf{x})(\varphi) \cong \varphi$, but the least fixed point Bot is everywhere undefined.

■ **Uniqueness of** Rec. If φ_{rec} is a recursion pfn, φ_{rec} is $\cong$-equivalent to Rec on pfnls.

$$\vartheta \in \mathit{Pfnl} \rightarrow \mathsf{Rec}(\vartheta) \cong \varphi_{\mathrm{rec}}(\vartheta)$$

This follows from the improved recursion theorem. More generally for any pfns $\varphi_{\mathrm{r},0}$ and $\varphi_{\mathrm{r},1}$ satisfying (rec.fix) and (rec.min) and any pfnl ϑ we have $\vartheta(\varphi_{\mathrm{r,j}}(\vartheta)) \cong \varphi_{\mathrm{r,j}}(\vartheta)$ for $j = 0,1$ by (rec.fix). $\varphi_{\mathrm{r},1}(\vartheta) \sqsubseteq \varphi_{\mathrm{r},0}(\vartheta)$ by (rec.min) for $\varphi_{\mathrm{r},1}$ and $\varphi_{\mathrm{r},0}(\vartheta) \sqsubseteq \varphi_{\mathrm{r},1}(\vartheta)$ by (rec.min) for $\varphi_{\mathrm{r},0}$. Hence $\varphi_{\mathrm{r},0}(\vartheta) \cong \varphi_{\mathrm{r},1}(\vartheta)$ as required.

■ **Parameterized Recursion.** In §IV.4 we showed how certain forms of recursion with parameters can described by treating the parameter as an argument or by carrying the parameter in the environment of a recursively defined pfn. This fact is an instance of a general result about the equivalence of alternative descriptions of recursion with parameters.

(param) $$\mathsf{Rec}(\lambda(\mathsf{h},\mathsf{x},\mathsf{z})\mathsf{g}(\mathsf{x},\mathsf{h}(\mathsf{x}),\mathsf{z})) \cong \lambda(\mathsf{x})\mathsf{Rec}(\lambda(\mathsf{f},\mathsf{z})\mathsf{g}(\mathsf{x},\mathsf{f},\mathsf{z}))$$

(param) is the *Rum* analog of equation (2.28) Scott [1976] and is proved in the same manner using the improved recursion theorem and the extensionality theorem.

Alternative description of recursion.

Our definition of Rec was derived by unwinding (i.e. extracting the lambda expression implicit in) Kleene's proof of the 2nd recursion theorem (Kleene [1952]: Theorem XXVII). An alternative "call-by-value" version of the Y combinator CbvY is given in Reynolds [1970].

▷ $$\mathsf{Y2} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{h})\lambda(\mathsf{x})\mathsf{h}(\mathsf{h},\mathsf{x})$$

▷ $$\mathsf{Y1} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{f})\lambda(\mathsf{h})\mathsf{f}(\mathsf{Y2}(\mathsf{h}))$$

▷ $$\mathsf{CbvY} \overset{\mathrm{df}}{\leftrightarrow} \lambda(\mathsf{f})\mathsf{Y1}(\mathsf{f},\mathsf{Y1}(\mathsf{f}))$$

■ For pfnls ϑ, $\Downarrow\mathsf{CbvY}'\vartheta$ and $\mathsf{CbvY}(\vartheta) \cong \mathsf{Rec}(\vartheta)$.

Proof: We show that CbvY is essentially a recursion pfn. Assume ϑ is a pfnl. We claim (i) $\mathsf{CbvY}'\vartheta \hookrightarrow \vartheta(\mathsf{Y2}(\mathsf{Y1}(\vartheta)))$ and (ii) $\lambda(\mathsf{f})\mathsf{Y2}(\mathsf{Y1}(\mathsf{f}))$ is a recursion pfn. Then by the improved recursion theorem and evaluation closure of $\cong$

$$\mathsf{CbvY}(\vartheta) \cong \vartheta(\mathsf{Y2}(\mathsf{Y1}(\vartheta))) \cong \mathsf{Y2}(\mathsf{Y1}(\vartheta)) \cong \mathsf{Rec}(\vartheta)$$

as required. (i) and (ii) follow directly from the definitions with a little computation. □

VI.2.3. Towards a *Rum* calculus

The maximum equivalence satisfies many laws expressed as an equational theory in the language of forms. These include the laws of the Lambda-v calculus (Plotkin [1975]). We list some of the laws for $\cong$ below and illustrate their use in proving equations by proving the equivalence of alternative descriptions of the until-loop functional.

■ **Laws for $\cong$.**

(if.if) $\mathsf{if}(\mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2),\mathfrak{f}_3,\mathfrak{f}_4) \cong \mathsf{if}(\mathfrak{f}_0,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathsf{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4))$

(if.app) $\mathfrak{f}(\mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2)) \cong \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}(\mathfrak{f}_1),\mathfrak{f}(\mathfrak{f}_2))$

(if.or) $\mathsf{if}(\mathsf{or}(\mathfrak{f}_0,\mathfrak{f}_1),\mathfrak{f}_2,\mathfrak{f}_3) \cong \mathsf{if}(\mathfrak{f}_0,\mathfrak{f}_2,\mathsf{if}(\mathfrak{f}_1,\mathfrak{f}_2,\mathfrak{f}_3))$

(let.cnv) $(\Downarrow\mathfrak{f}|^{s}_{\mathfrak{f}_0} \rightarrow \Downarrow\mathfrak{f}_0) \rightarrow \mathsf{let}\{s \leftarrow \mathfrak{f}_0\}\mathfrak{f} \cong \mathfrak{f}|^{s}_{\mathfrak{f}_0}$

(let.perm) $\mathsf{let}\{s_0 \leftarrow \mathsf{let}\{s_1 \leftarrow \mathfrak{f}_1\}\mathfrak{f}_0\}\mathfrak{f} \cong \mathsf{let}\{s_1 \leftarrow \mathfrak{f}_1\}\mathsf{let}\{s_0 \leftarrow \mathfrak{f}_0\}\mathfrak{f}$

% s_1 not free in $\mathfrak{f}$

(cart.id) $\mathsf{cart}(\mathfrak{f},\mathsf{mt}) \cong \mathsf{cart}(\mathsf{mt},\mathfrak{f}) \cong \mathfrak{f}$

(cart.assoc) $\mathsf{cart}(\mathsf{cart}(\mathfrak{f}_0,\mathfrak{f}_1),\mathfrak{f}_2) \cong \mathsf{cart}(\mathfrak{f}_0,\mathsf{cart}(\mathfrak{f}_1,\mathfrak{f}_2))$

(subst) $\mathfrak{f}_0 \cong \mathfrak{f}_1 \rightarrow \mathfrak{f}|^{s}_{\mathfrak{f}_0} \cong \mathfrak{f}|^{s}_{\mathfrak{f}_1}$

(abstract) $\mathfrak{f}_0 \cong \mathfrak{f}_1 \rightarrow \lambda(s)\mathfrak{f}_0 \cong \lambda(s)\mathfrak{f}_1$ % For s not a global symbol

(rec.def) $s_{\mathfrak{f}} \cong \mathsf{Rec}(\lambda(s_{\mathfrak{f}})\lambda(s_1 \ldots s_n)\mathfrak{f}_{\mathrm{body}}) \rightarrow s_{\mathfrak{f}}(s_1 \ldots s_n) \cong \mathfrak{f}_{\mathrm{body}}$

% For $s_1, \ldots s_n$ not global symbols

(param) $\mathsf{Rec}(\lambda(\mathsf{h},\mathsf{x},\mathsf{z})\mathsf{g}(\mathsf{x},\mathsf{h}(\mathsf{x}),\mathsf{z})) \cong \lambda(\mathsf{x})\mathsf{Rec}(\lambda(\mathsf{f},\mathsf{z})\mathsf{g}(\mathsf{x},\mathsf{f},\mathsf{z}))$

The laws involving the recursion pfn have already been discussed. (abstract) is just an alternative form of (ext). The remaining laws are consequences of general closure properties of $\cong$ which will be proved in §4. We call the use of (let.cnv) to replace an instance of the left-hand side by an instance of the right-hand side let-elimination. Use of (let.cnv) in the opposite sense is called let-introduction.

Alternative until-loop constructions.

To illustrate the use of the parameterization law we derive the equivalence of alternative descriptions of the until-loop functional Until and Un.

▷ $\mathsf{Until} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{p},\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{un})\lambda(\mathsf{s})\mathsf{if}(\mathsf{p}(\mathsf{s}),\mathsf{s},\mathsf{un}(\mathsf{f}(\mathsf{s}))))$

▷ $\mathsf{Un} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{Un})\lambda(\mathsf{p},\mathsf{f},\mathsf{s})\mathsf{if}(\mathsf{p}(\mathsf{s}),\mathsf{s},\mathsf{Un}(\mathsf{p},\mathsf{f},\mathsf{f}(\mathsf{s}))))$

Beginning with the definition of Un, let-introduction is used to put the body of the definition in a format that matches the (param) law. The (param) law is then applied followed by let-elimination to recover the original definition of Until.

$$
\begin{aligned}
&\mathsf{Rec}(\lambda(\mathsf{Un})\lambda(\mathsf{p},\mathsf{f},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{x},\mathsf{Un}(\mathsf{p},\mathsf{f},\mathsf{f}(\mathsf{x}))))\\
&\quad\cong \mathsf{Rec}(\lambda(\mathsf{Un})\lambda(\mathsf{p},\mathsf{f},\mathsf{x})\{\lambda(\mathsf{p},\mathsf{f},\mathsf{h},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{x},\mathsf{h}(\mathsf{p},\mathsf{f}(\mathsf{x})))\}(\mathsf{p},\mathsf{f},\mathsf{Un}(\mathsf{p},\mathsf{f}),\mathsf{x}))\\
&\qquad \text{\% let introduction}\\
&\quad\cong \lambda(\mathsf{p},\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{h})\lambda(\mathsf{x})\{\lambda(\mathsf{p},\mathsf{f},\mathsf{h},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{x},\mathsf{h}(\mathsf{f}(\mathsf{x})))\}(\mathsf{p},\mathsf{f},\mathsf{h},\mathsf{x}))\\
&\qquad \text{\% (param) with } \mathsf{g} = \lambda(\mathsf{p},\mathsf{f},\mathsf{h},\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{x},\mathsf{h}(\mathsf{p},\mathsf{f}(\mathsf{x})))\\
&\quad\cong \lambda(\mathsf{p},\mathsf{f})\mathsf{Rec}(\lambda(\mathsf{un})\lambda(\mathsf{x})\mathsf{if}(\mathsf{p}(\mathsf{x}),\mathsf{x},\mathsf{un}(\mathsf{f}(\mathsf{x}))))\\
&\qquad \text{\% let elimination}
\end{aligned}
$$

The laws for $\cong$ (with some additional laws for if-simplification and conditional substitution) can also be used to formalize the derivation of the equation for Tprodc (see §II.3).

VI.2.4. Remarks

Towards a theory of program transformations. With respect to developing a theory of program transformations, the laws for $\cong$ give rise to a rich collection of transformation rules. What is needed is to develop further principles such as laws for safe introduction of recursion. One approach is to try to extend the notion of computational progress (Scherlis [1980]) to *Rum* in order to obtain safe systems of rules for deriving recursive definitions.

Another direction of work is to look for extensions of the Lambda-v calculus (Plotkin [1975]) by adding laws for if, mt, cart, fst, rst. Such added laws should preserve the Church-Rosser property and reflect adequately the intended interpretation of these additional primitives. This would provide a richer base of "syntactically" generated equivalence relations and be a step toward mechanizing transformations or the checking of validity of transformation steps.

A third direction of work is to explore the use of comparisons other than $\sqsubseteq$ and $\cong$ to formulate soundness of transformations as well as to express forms of improvement made by transformations.

Open Questions. We have really just exposed the tip of the iceberg. There are a number of questions still to be answered about the maximum comparisons. Some of the open questions are:

- What is the logical complexity of $\sqsubseteq$ (or $\cong$)?

- Is $\sqsubseteq$ the least comparison satisfying extensionality?
- Are the fixed points computed by Rec the only uniformly computable fixed points of pfnls in $\mathcal{R}um$? That is, is there a pfnl φ_{fix} such that for any pfnl ϑ $\varphi_{\mathrm{fix}}(\vartheta) \cong \vartheta(\varphi_{\mathrm{fix}}(\vartheta))$ but for some pfnl ϑ we have $\varphi_{\mathrm{fix}}(\vartheta) \not\cong \mathsf{Rec}(\vartheta)$).
- $\sqsubseteq$ is defined as the maximum approximation for a fixed but arbitrary data structure $\mathfrak{D}$. Let $\sqsubseteq_{\emptyset}$ be the restriction of $\sqsubseteq$ to the pure dtrees - dtrees that have no occurrences of data or data operations. Is this relation the same for all choices of $\mathfrak{D}$? If not, how does it depend on $\mathfrak{D}$?

VI.3. Pfn extensions of comparisons

Pfn extensions provide a method of extending a given comparison to a comparison containing a given pfn relation. This in turn provides a method of proving that a given pfn relation is contained in the maximum approximation. Since for these purposes only the restriction to pfns of a dtree relation is relevant, we introduce the notion of *pfn comparison.* This characterizes the relations obtained by restricting dtree comparisons to pfns. We define a *pfn extension operation* and a *pfn extension condition* which guarantees that the pfn extension of a pfn comparison is a pfn comparison. These tools are used to prove the extensionality and improved recursion theorems stated in §2. We also prove the uniqueness of Self, thus answering one of the questions asked at the beginning of this chapter. Originally, these and other theorems of similar form were proved individually. It then became clear that these proofs were all based on the same extension construction and that the proofs amounted to verifying the pfn extension condition. We shall not make further attempt to motivate the definitions. The proof of the pfn extension theorem should help make the technical reasons for the definitions clearer.

VI.3.1. Pfn comparisons

▷ **Pfn comparison.** A *pfn comparison* is a dtree relation that is reflexive, transitive and is evaluation closed under application (satisfies C^p). $\mathcal{C}_{\mathbf{P}}$ is the set of pfn comparisons.

$$C^p(\rho) \underset{\mathrm{df}}{\equiv} (\forall \varphi_0, \varphi_1, v_0, v_1)(\varphi_0 \, \rho \, \varphi_1 \wedge v_0 \, \rho \, v_1 \rightarrow (\varphi_0{}' v_0) \, \overline{\rho} \, (\varphi_1{}' v_1))$$

$$\rho \in \mathcal{C}_{\mathbf{P}} \underset{\mathrm{df}}{\equiv} \rho = \rho^* \wedge C^p(\rho)$$

Note that for a pfn comparison, there are no requirements for dtrees other than pfns. In fact, pfn comparisons are essentially dtree comparisons with non-pfn dtrees forgotten. This is expressed by the pfn comparison lemma.

▪ **Pfn comparison lemma.** Each dtree comparison is a pfn comparison and the evaluation closure of a pfn comparison is a (dtree) comparison containing the restriction to pfns of the given pfn comparison.

$$\rho \in \mathcal{C} \rightarrow \rho \in \mathcal{C}_{\mathbf{P}}$$

$$\rho \in \mathcal{C}_{\mathbf{P}} \rightarrow \rho[\mathbb{P}] \subset \overline{\rho} \in \mathcal{C}$$

VI.3.2. The pfn extension theorem

The pfn extension of a pfn comparison by a pfn relation is the least transitive reflexive relation containing the union of the given relations and closed under value substitution. For technical reasons we construct the extension in two stages.

▹ **Value closure.** The value closure $\chi^{\sharp}$ of a relation χ is the least reflexive relation such that[1]

(0) $\chi[\mathbb{P}] \subset \chi^{\sharp}$

(1) $u_{0,1}\ \chi^{\sharp}\ u_{1,1} \wedge \ldots \wedge u_{0,n}\ \chi^{\sharp}\ u_{1,n} \rightarrow \varphi\{\triangleleft u_{0,1} \ldots \triangleleft u_{0,n}\}\ \chi^{\sharp}\ \varphi\{\triangleleft u_{1,1} \ldots \triangleleft u_{1,n}\}$

▹ **Pfn extension.** The pfn extension of ρ by χ, (written $\rho\sharp\chi$), is the transitive union of ρ and the value closure of χ.

$$\rho\sharp\chi \underset{\text{df}}{=} \uplus\{\rho, \chi^{\sharp}\}$$

In order to state the pfn extension condition we introduce two more definitions: relative pfn comparison and the dtree relation induced by the value closure of a pfn relation.

▹ **Relative pfn comparison.** We say that χ is a pfn comparison relative to ρ if (χ, ρ) satisfies C_r^p where

$$C_r^p(\chi, \rho) \underset{\text{df}}{\equiv} (\forall \varphi_0, \varphi_1)(\forall u_0, u_1)(\varphi_0\ \chi\ \varphi_1 \wedge u_0\ \chi\ u_1 \rightarrow (\varphi_0\,{}'\,u_0)\ \overline{\chi} \circ \overline{\rho}\ (\varphi_1\,{}'\,u_1))$$

Recall that $\circ$ is the composition operation and $\eth_0\ \rho_0 \circ \rho_1\ \eth_1 \leftrightarrow (\exists \eth)(\eth_0\ \rho_0\ \eth\ \rho_1\ \eth_1)$. The interest in relative pfn comparisons is explained by the following lemma.

[1] The requirement that $\chi^{\sharp}$ be reflexive amounts to adding a further clause $\rho_{=}[\mathbf{P}] \subset \chi^{\sharp}$. Since this clause corresponds to a trivial case in the arguments based on the definition of $\chi^{\sharp}$ we have not listed it explicitly.

■ **Relative pfn comparison lemma.** If ρ is a pfn comparison and χ is a pfn comparison relative to ρ then $\biguplus\{\rho,\chi\}$ is a pfn comparison.

$$\rho \in \mathcal{C}_{\mathbf{P}} \wedge \mathcal{C}^{p}_{r}(\chi,\rho) \rightarrow \biguplus\{\rho,\chi\} \in \mathcal{C}_{\mathbf{P}}$$

This follows from $C^{p}(\rho) \wedge C^{p}_{r}(\chi,\rho) \rightarrow C^{p}(\biguplus\{\rho,\chi\})$ which is easy to check by unwinding the definitions.

▷ **Value closure on dtrees.** The dtree relation $\chi^{\sharp,d}$ induced by the value closure of a relation χ is the least dtree relation such that

(0) $\partial_0\ \chi[\mathbb{P}]\ \partial_1 \rightarrow \partial_0\ \chi^{\sharp,d}\ \partial_1$

(1) $u_{0,1}\ \chi^{\sharp}\ u_{1,1} \wedge \ldots \wedge u_{0,n}\ \chi^{\sharp}\ u_{1,n} \rightarrow \partial\{\triangleleft u_{0,1}\ldots\triangleleft u_{0,n}\}\ \chi^{\sharp,d}\ \partial\{\triangleleft u_{1,1}\ldots\triangleleft u_{1,n}\}$

(2) $\varphi_0\ \chi^{\sharp}\ \varphi_1 \wedge u_0\ \chi^{\sharp}\ u_1 \rightarrow (\varphi_0{}'u_0)\ \chi^{\sharp,d}\ (\varphi_1{}'u_1)$

We are now ready to define the pfn extension condition Θ^{p} and to prove the pfn extension theorem.

▷ **Pfn extension condition.**

$$\Theta^{p}(\chi,\rho) \underset{\text{df}}{\equiv} (\forall\varphi_0,\varphi_1,u_0,u_1)(\varphi_0\ \chi\ \varphi_1 \wedge u_0\ \chi^{\sharp}\ u_1 \wedge \Theta^{p}_{hyp}(\chi,\rho,\varphi_0{}'u_0)$$
$$\rightarrow \quad (\varphi_0{}'u_0)\ \overline{\chi^{\sharp}} \circ \bar{\rho}\ (\varphi_1{}'u_1))$$

where

$$\Theta^{p}_{hyp}(\chi,\rho,\partial_0) \underset{\text{df}}{\equiv} (\forall\partial_{a} \prec \partial_0)(\forall\partial_{b})(\partial_{a}\ \chi^{\sharp,d}\ \partial_{b} \rightarrow \partial_{a}\ \overline{\chi^{\sharp}} \circ \bar{\rho}\ \partial_{b})$$

■ **Pfn extension theorem.** If ρ is a pfn comparison and (χ,ρ) satisfies the pfn extension condition then the pfn extension of ρ by χ is a pfn comparison.

$$\rho \in \mathcal{C}_{\mathbf{P}} \wedge \Theta^{p}(\chi,\rho) \rightarrow \rho\sharp\chi \in \mathcal{C}_{\mathbf{P}}$$

Before proving the pfn extension theorem we prove two lemmas. The theorem will then follow easily.

■ **Lemma.1.** If ρ is a pfn comparison, $\Theta^{p}_{hyp}(\chi,\rho,\partial_0)$, $u_{0,i}\ \chi^{\sharp}\ u_{1,i}$ for $1 \leq i \leq n$, and $\partial_{j} = \partial\{\triangleleft u_{j,1}\ldots\triangleleft u_{j,n}\}$ for $j = 0,1$ then $\partial_0\ \overline{\chi^{\sharp}} \circ \bar{\rho}\ \partial_1$.

Proof: Lemma.1. Assume $\partial_0 \hookrightarrow v_0$ and show

$$(\exists v_2,v_1)(v_0\ \chi^{\sharp}\ v_2\ \rho\ v_1 \wedge \partial_1 \hookrightarrow v_1)$$

. We consider cases according to the dtree context construction.

Case($\mathfrak{d}_{\mathrm{j}} = \mathsf{Mt}$): let $v_0 = v_1 = v_2 = \Box$.

Case($\mathfrak{d}_{\mathrm{j}} = \triangleleft u_{\mathrm{j}}$): let $v_0 = u_0$, $v_1 = v_2 = u_1$.

Case($\mathfrak{d}_{\mathrm{j}} = \varphi_{\mathrm{a,j}}$): let $v_0 = \mathfrak{d}_0$, $v_1 = v_2 = \mathfrak{d}_1$.

Case($\mathfrak{d}_{\mathrm{j}} = \mathsf{app}(\mathfrak{d}_{\mathrm{a,j}}, \mathfrak{d}_{\mathrm{b,j}})$): By definition of $\chi^{\sharp,d}$ we have $\mathfrak{d}_{\mathrm{a},0}\ \chi^{\sharp,d}\ \mathfrak{d}_{\mathrm{a},1}$ and $\mathfrak{d}_{\mathrm{b},0}\ \chi^{\sharp,d}\ \mathfrak{d}_{\mathrm{b},1}$. By Θ^{p}_{hyp} and properties of evaluation and subcomputation, there are $u_{\mathrm{a,i}}$ and $u_{\mathrm{b,i}}$ for $i = 0, 1, 2$ such that

$$\mathfrak{d}_{\mathrm{a},0} \hookrightarrow u_{\mathrm{a},0} \quad \text{and} \quad \mathfrak{d}_{\mathrm{b},0} \hookrightarrow u_{\mathrm{b},0} \quad \text{and} \quad u_{\mathrm{a},0}{}' u_{\mathrm{b},0} \hookrightarrow v_0,$$

$$\mathfrak{d}_{\mathrm{a},1} \hookrightarrow u_{\mathrm{a},1} \quad \text{and} \quad \mathfrak{d}_{\mathrm{b},1} \hookrightarrow u_{\mathrm{b},1},$$

$$u_{\mathrm{a},0}\ \chi^{\sharp}\ u_{\mathrm{a},2}\ \rho\ u_{\mathrm{a},1} \quad \text{and} \quad u_{\mathrm{b},0}\ \chi^{\sharp}\ u_{\mathrm{b},2}\ \rho\ u_{\mathrm{b},1}.$$

Since the application is defined we know that $u_{\mathrm{a},0} \in \mathbb{O} \wedge u_{\mathrm{b},0} \in \mathbb{D}^*$ or $u_{\mathrm{a},0} \in \mathbb{P}$. In the case $u_{\mathrm{a},0} \in \mathbb{O}$, by definition of the extension of a dtree relation to values, $u_{\mathrm{a},0} = u_{\mathrm{a},2} = u_{\mathrm{a},1}$ and $u_{\mathrm{b},0} = u_{\mathrm{b},2} = u_{\mathrm{b},1}$ and we let $v_2 = v_1 = v_0$. In the case $u_{\mathrm{a},0} \in \mathbb{P}$ we have

$$(u_{\mathrm{a},0}{}' u_{\mathrm{b},0})\ \chi^{\sharp,d}\ (u_{\mathrm{a},2}{}' u_{\mathrm{b},2})\ \rho\ (u_{\mathrm{a},1}{}' u_{\mathrm{b},1})$$

so by Θ^{p}_{hyp} there are v_2 and v_3 such that $v_0\ \chi^{\sharp}\ v_2\ \rho\ v_3$ and $u_{\mathrm{a},2}{}' u_{\mathrm{b},2} \hookrightarrow v_3$. Since ρ is a comparison there is v_1 such that $v_3\ \rho\ v_1$ and $u_{\mathrm{a},1}{}' u_{\mathrm{b},1} \hookrightarrow v_1$. The if-, cart-, fst-, and rst- cases are similar. $\Box_{lemma.1}$

■ **Lemma.2.**

$$\Theta^{p}(\chi, \rho) \wedge \rho \in \mathcal{C}_{\mathbf{P}} \rightarrow (\mathfrak{d}_0\ \chi^{\sharp,d}\ \mathfrak{d}_1 \rightarrow \mathfrak{d}_0\ \overline{\chi^{\sharp}} \circ \overline{\rho}\ \mathfrak{d}_1)$$

Proof: Lemma.2. We use computation induction on $\mathfrak{d}_0$, with induction hypothesis $\Theta^{p}_{hyp}(\chi, \rho, \mathfrak{d}_0)$ and consider cases according to the definition of $\chi^{\sharp,d}$.

Case.0: $\mathfrak{d}_0\ \chi[\mathbb{P}]\ \mathfrak{d}_1$. Then $\mathfrak{d}_0\ \overline{\chi^{\sharp}}\ \mathfrak{d}_1$ since $\mathfrak{d}_{\mathrm{j}} \hookrightarrow \mathfrak{d}_{\mathrm{j}}$.

Case.1: $\mathfrak{d}_{\mathrm{j}} = \mathfrak{d}\{\triangleleft u_{\mathrm{j},1} \ldots \triangleleft u_{\mathrm{j,n}}\}$, for $j = 0, 1$ where $u_{0,\mathrm{i}}\ \chi^{\sharp}\ u_{1,\mathrm{i}}$ for $1 \leq i \leq n$. This follows from Lemma.1 using the induction hypothesis.

Case.2: $\mathfrak{d}_0 = \varphi_0{}' u_0$ and $\mathfrak{d}_1 = \varphi_1{}' u_1$ where $\varphi_0\ \chi^{\sharp}\ \varphi_1$ and $u_0\ \chi^{\sharp}\ u_1$. Using the definition of $\chi^{\sharp}$, there are two subcases.

Case.2.0: $\varphi_0\ \chi\ \varphi_1$. This follows from Θ^{p} using the induction hypothesis.

Case.2.1: $\varphi_{\mathrm{j}} = \varphi\{\triangleleft u_{\mathrm{j},1} \ldots \triangleleft u_{\mathrm{j,n}}\}$ for $j = 0, 1$ with $u_{0,\mathrm{i}}\ \chi^{\sharp}\ u_{1,\mathrm{i}}$ for $1 \leq i \leq n$. This is just Case.1. $\Box_{lemma.2}$

Proof: Pfn extension theorem. As a special case of lemma.2 we have

$$\rho \in \mathcal{C}_{\mathbf{P}} \wedge \Theta^{p}(\chi, \rho) \rightarrow C^{p}_{r}(\chi^{\sharp}, \rho)$$

Using the relative pfn comparison lemma and $\rho \sharp \chi = \uplus\{\rho, \chi^{\sharp}\}$ it follows that $\rho \sharp \chi \in \mathcal{C}_{\mathbf{P}}$ as required. $\Box_{p.e.t}$

VI.3.3. Using the pfn extension theorem

Our applications of the pfn comparison theorem will be to show that certain pfn relations are contained in the maximum comparisons. These are based on the following corollary.

■ **Pfn extension corollary.** To prove that a pfn relation χ is contained in $\sqsubseteq$ or to prove that a symmetric pfn relation χ is contained in $\cong$ it suffices to show that $\Theta^p(\chi, \sqsubseteq)$.

$$\Theta^p(\chi, \sqsubseteq) \rightarrow \chi[\mathbb{P}] \subset \sqsubseteq$$

$$\Theta^p(\chi, \sqsubseteq) \wedge \chi = \chi^- \rightarrow \chi[\mathbb{P}] \subset \cong$$

Proof: By the pfn comparison lemma and the pfn extension theorem and maximality of $\sqsubseteq$, if $\Theta^p(\chi, \sqsubseteq)$ then $\chi[\mathbb{P}] \subset \overline{\sqsubseteq \sharp \chi} \subset \sqsubseteq$ and by definition of $\cong$, $(\chi \cap \chi^-)[\mathbb{P}] \subset \cong$.

• **Proof of the extensionality theorem.** For the first application, we prove the extensionality of $\sqsubseteq$.

$$(\text{ext.}\sqsubseteq) \qquad (\forall v)(\varphi_0{}'v \sqsubseteq \varphi_1{}'v) \leftrightarrow \varphi_0 \sqsubseteq \varphi_1)$$

Let $\chi = \{(\varphi_0, \varphi_1) \mid (\forall v)\varphi_0{}'v \sqsubseteq \varphi_1{}'v\}$. Since $\sqsubseteq \in \mathcal{C}$ the (only-if) direction of (ext.$\sqsubseteq$) is trivial, so we need only show the (if) direction – $\chi \subset \sqsubseteq$. By the pfn extension corollary it suffices to show $\Theta^p(\chi, \sqsubseteq)$. To see this assume $\varphi_0\ \chi\ \varphi_1$ and $u_0\ \chi^\sharp\ u_1$ and $\Theta^p_{hyp}(\chi, \sqsubseteq, \varphi_0{}'u_0)$. Then by definition of χ and of value closure, $(\varphi_0{}'u_0)\ \chi^\sharp\ (\varphi_0{}'u_1) \sqsubseteq (\varphi_1{}'u_1)$. By lemma.1 $(\varphi_0{}'u_0)\ \overline{\chi^\sharp} \circ \overline{\sqsubseteq}\ (\varphi_1{}'u_1)$, as required. $\square_{ext}$

• **Proof of the improved recursion theorem.** Our second application of the pfn comparison theorem is to prove the main part of the improved recursion theorem

$$(\text{rec.min}) \qquad \vartheta(\varphi) \sqsubseteq \varphi \rightarrow \varphi_{\mathrm{r}}(\vartheta) \sqsubseteq \varphi$$

where φ_{r} is a recursion pfn (see §2) and ϑ is a pfnl. Assume $\vartheta(\varphi) \sqsubseteq \varphi$ and let $\chi = \{(\varphi_{\mathrm{r}}(\vartheta), \varphi)\}$. By the pfn extension corollary to show $\varphi_{\mathrm{r}}(\vartheta) \sqsubseteq \varphi$. we need only verify $\Theta^p(\chi, \sqsubseteq)$. Assume $\varphi_0\ \chi\ \varphi_1$, $u_0\ \chi^\sharp\ u_1$ and $\Theta^p_{hyp}(\chi, \sqsubseteq, \varphi_0{}'u_0)$. By definition of χ, $\varphi_0 = \varphi_{\mathrm{r}}(\vartheta)$ and $\varphi_1 = \varphi$. Assume $\varphi_0{}'u_0 \hookrightarrow v_0$, then by properties of recursion pfns, $\vartheta{}'\varphi_0 \prec \varphi_0{}'u_0$ and $\varphi_0{}'u_0 \ggcurly \vartheta(\varphi_0){}'u_0$. By Θ^p_{hyp} and the assumptions on φ we find φ_2 with $\vartheta(\varphi_0)\ \chi^\sharp\ \varphi_2 \sqsubseteq \vartheta(\varphi_1) \sqsubseteq \varphi_1$ thus

$$(\varphi_0{}'u_0) \rightleftharpoons (\vartheta(\varphi_0){}'u_0)\ \chi^{\sharp,d}\ (\varphi_2{}'u_1) \sqsubseteq (\varphi_1{}'u_1)$$

and by Θ^{p}_{hyp} we have

$$(\vartheta(\varphi_0)'u_0)\ \overline{\chi^\sharp} \circ \sqsubseteq\ (\varphi_2'u_1) \sqsubseteq (\varphi_1'u_1)$$

as required. $\Box_{(rec.min)}$

• **Proof of the uniqueness of self.** To answer the question raised at the beginning of this chapter we prove that if $\varphi' v \hookrightarrow \varphi$ for all v then $\varphi \cong \mathsf{Self}$. Let

$$\chi = \{(\varphi_0, \varphi_1) \mid (\forall v)(\varphi_0' v \hookrightarrow \varphi_0 \wedge \varphi_1' v \hookrightarrow \varphi_1)\}.$$

By the pfn extension corollary, since χ is symmetric, to show $\chi \subset \cong$ we need only verify $\Theta^{p}(\chi, \sqsubseteq)$. Assume $\varphi_0\ \chi\ \varphi_1$ and $u_0\ \chi^\sharp\ u_1$. By definition of χ, $\varphi_0' u_0 \hookrightarrow \varphi_0$ and $\varphi_1' u_1 \hookrightarrow \varphi_1$ thus $(\varphi_0' u_0)\ \overline{\chi^\sharp}\ (\varphi_1' u_1)$. $\Box_{self}$

Exercise: Stream equivalence. Define the restriction operation on streams StrR which restricts the inputs to a stream to be the empty sequence

$$\triangleright \qquad \mathsf{StrR} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{s})\lambda(\mathsf{z})\mathsf{let}\{[\mathsf{x},\mathsf{s}] \leftarrow \mathsf{s}()\}\mathsf{ifmt}(\mathsf{x},\mathsf{mt},[\mathsf{x},\mathsf{StrR}(\mathsf{s})])$$

Prove that if two streams ϑ_0, ϑ_1 are equal as streams then the restrictions of the two streams are $\cong$-equivalent. For streams ϑ_0,ϑ_1

$$\vartheta_0 \overset{s}{=} \vartheta_1 \rightarrow \mathsf{StrR}(\vartheta_0) \cong \mathsf{StrR}(\vartheta_1)$$

(See §IV.6 for definitions and properties of streams)

Hint: Define the pfn extension of $\cong$ by

$$\{(\vartheta_0, \vartheta_1)\} \cup \{(\vartheta_0^{>n}, \vartheta_1^{>n}) \mid n < |\vartheta_0|_s\}$$

and use the pfn extension theorem.

Remark: We need to work with the restriction here because we have been discussing streams in terms of repeated application to the empty sequence. Thus it is really the restriction that we are interested in.

VI.4. Dtree extensions of comparisons

Dtree extensions provide a means of extending a given comparison to a comparison containing a given dtree relation. The main interest is in obtaining relations that contain sets of program transformations. Thus we focus on substitution-closed relations. The basic plan is similar to that for pfn extensions. We define a *dtree extension operation* and a *dtree extension condition* which guarantees that the dtree extension of a comparison is a comparison. The dtree extension operation is used to construct comparisons satisfying laws such as those given for $\cong$ given in §2. These constructions also serve to prove the corresponding laws for $\cong$. A one step extensionality closure operation is defined which provides a means of obtaining comparisons approximating extensionality to any desired level. This also serves as an alternative proof of the extensionality theorem. As for pfn extensions, the basic definitions were derived by abstracting on a collection of proofs based on the dtree extension construction and which amounted to verifying the dtree extension condition.

VI.4.1. The dtree extension operation

The dtree extension of a relation ρ by a relation χ is constructed in two stages. First form the dtree substitution closure of χ, then take the transitive union with ρ.

▹ **Substitution closure.** The substitution closure $\chi^{\natural}$ of χ is the least reflexive dtree relation such that

(0) $\chi \subset \chi^{\natural}$

(1) $u_0 \, \chi^{\natural} \, u_1 \to \triangleleft u_0 \, \chi^{\natural} \, \triangleleft u_1$

(2) $\mathfrak{d}_{0,1} \, \chi^{\natural} \, \mathfrak{d}_{1,1} \ldots \mathfrak{d}_{0,n} \, \chi^{\natural} \, \mathfrak{d}_{1,n} \to \mathfrak{d}\{\mathfrak{d}_{0,1} \ldots \mathfrak{d}_{0,n}\} \, \chi^{\natural} \, \mathfrak{d}\{\mathfrak{d}_{1,1} \ldots \mathfrak{d}_{1,n}\}$

In clause (2) we may assume with out loss that $\mathfrak{d}\{\ldots\}$ is not a trivial context i.e. that $\mathfrak{d}\{\ldots\}$ is not simply a hole.

▹ **Dtree extension.** The dtree extension $\rho \natural \chi$ of ρ by χ is defined by

$$\rho \natural \chi \underset{\mathrm{df}}{=} \uplus\{\rho, \chi^{\natural}\}$$

■ **Dtree extension lemma.** The substitution closure of a relation ρ is substitution-closed and dtree extension commutes with inversion.

$$Cl_{dsubst}(\rho^{\natural}) \quad \text{and} \quad (\rho \natural \chi)^{-} = (\rho^{-} \natural \chi^{-})$$

This is an easy consequences of the definitions and properties of substitution-closure.

VI.4.2. The dtree extension theorem

We now define the dtree extension condition for relations (χ, ρ) and prove that it guarantees that the dtree extension of a comparison ρ by a relation χ is a comparison. As for pfn extensions, the dtree extension condition is expressed in terms of relative comparisons.

▹ **Relative comparison.** We say that χ is a comparison relative to ρ if (χ, ρ) satisfy C_r^e and C_r^a where

$$C_r^e(\chi,\rho) \underset{\text{df}}{\equiv} (\forall \mathfrak{d}_0, \mathfrak{d}_1)(\mathfrak{d}_0 \, \chi \, \mathfrak{d}_1 \rightarrow \mathfrak{d}_0 \, \overline{\chi} \circ \overline{\rho} \, \mathfrak{d}_1)$$

$$C_r^a(\chi,\rho) \underset{\text{df}}{\equiv} (\forall \varphi_0, \varphi_1)(\forall u_0, u_1)(\varphi_0 \, \chi \, \varphi_1 \wedge u_0 \, \chi \, u_1 \rightarrow (\varphi_0{}' u_0) \, \chi \circ \rho \, (\varphi_1{}' u_1))$$

The key fact about relative comparisons is the following.

▪ **Relative Comparison Lemma.** If ρ is a comparison and χ is a comparison relative to ρ then $\uplus\{\rho, \chi\}$ is a comparison.

$$\rho \in C \wedge C_r^e(\chi,\rho) \wedge C_r^a(\chi,\rho) \rightarrow \uplus\{\rho,\chi\} \in C$$

This follows from

$$C^e(\rho) \wedge C_r^e(\chi,\rho) \rightarrow C^e(\uplus\{\rho,\chi\})$$

$$C^a(\rho) \wedge C_r^a(\chi,\rho) \rightarrow C^a(\uplus\{\rho,\chi\})$$

which are easy to check by unwinding the definitions.

▹ **Dtree extension condition.** The dtree extension condition $\Theta(\chi, \rho)$ for (χ, ρ) is defined by

$$\Theta(\chi,\rho) \underset{\text{df}}{\equiv} \Theta^e(\chi,\rho) \wedge \Theta^a(\chi,\rho)$$

where

$$\Theta^e(\chi,\rho) \underset{\text{df}}{\equiv} (\forall \mathfrak{d}_0, \mathfrak{d}_1)(\mathfrak{d}_0 \, \chi \, \mathfrak{d}_1 \wedge \Theta^e_{hyp}(\chi,\rho,\mathfrak{d}_0) \rightarrow \mathfrak{d}_0 \; \overline{\chi^\natural} \circ \overline{\rho} \; \mathfrak{d}_1)$$

$$\Theta^e_{hyp}(\chi,\rho,\mathfrak{d}_0) \underset{\text{df}}{\equiv} (\forall \mathfrak{d}_a \prec \mathfrak{d}_0)(\forall \mathfrak{d}_b)(\mathfrak{d}_a \, \chi^\natural \, \mathfrak{d}_b \rightarrow \mathfrak{d}_a \; \overline{\chi^\natural} \circ \overline{\rho} \; \mathfrak{d}_b)$$

$$\Theta^a(\chi,\rho) \underset{\text{df}}{\equiv} (\forall \varphi_0, \varphi_1, u_0, u_1)(\varphi_0 \, \chi \, \varphi_1 \wedge u_0 \, \chi^\natural \, u_1 \rightarrow (\varphi_0{}' u_0) \, \chi^\natural \circ \rho \, (\varphi_1{}' u_1))$$

▪ **Dtree extension theorem.** If ρ is a comparison and (χ, ρ) satisfies the dtree extension condition then the dtree extension of ρ by χ is a comparison.

$$\rho \in C \wedge \Theta(\chi,\rho) \rightarrow \rho \natural \chi \in C$$

Proof: Dtree extension theorem. We show

(claim.a) $$\Theta^a(\chi,\rho) \to C_r^a(\chi^\natural,\rho)$$

and

(claim.e) $$\Theta(\chi,\rho) \land \rho \in \mathcal{C} \to C_r^e(\chi^\natural,\rho)$$

then $\Theta(\chi,\rho) \land \rho \in \mathcal{C} \to \rho\natural\chi \in \mathcal{C}$ follows from the relative comparison lemma and the definition of $\rho\natural\chi$.

Proof (claim.a.): Assume $\Theta^a(\chi,\rho)$, $\varphi_0\ \chi^\natural\ \varphi_1$, and $u_0\ \chi^\natural\ u_1$ and show

$$(\varphi_0{}'u_0)\ \chi^\natural \circ \rho\ (\varphi_1{}'u_1).$$

According to the definition of $\chi^\natural$ there are three cases.

(Case.0): $\varphi_0\ \chi\ \varphi_1$. Here $(\varphi_0{}'u_0)\ \chi^\natural \circ \rho\ (\varphi_1{}'u_1)$ follows from $\Theta^a(\chi,\rho)$.

(Case.1): $\varphi_j = \lhd u_j$. This case is impossible as a value dtree can not be a pfn.

(Case.2): $\varphi_j = \varphi\{\ldots\mathfrak{d}_{j,i}\ldots\}$ where $\mathfrak{d}_{0,i}\ \chi^\natural\ \mathfrak{d}_{1,i}$. Here $(\varphi_0{}'u_0)\ \chi^\natural\ (\varphi_1{}'u_1)$ by the definition of $\chi^\natural$ since $\varphi\{\ldots\mathfrak{d}_{j,i}\ldots\}{}'u_j = \varphi\{\lhd u_j\ldots\mathfrak{d}_{j,i}\ldots\}{}'u_j$.

$\Box_{claim.a}$

Proof (claim.e.): Assume $\Theta^e(\chi,\rho)$ and $\rho \in \mathcal{C}$ and show by computation induction on $\mathfrak{d}_0$ that

$$\mathfrak{d}_0\ \chi^\natural\ \mathfrak{d}_1 \to \mathfrak{d}_0\ \overline{\chi^\natural} \circ \overline{\rho}\ \mathfrak{d}_1$$

Note that the induction hypothesis is $\Theta^e_{hyp}(\chi,\rho,\mathfrak{d}_0)$. By the definition of $\chi^\natural$, there are three cases.

(Case.0): $\mathfrak{d}_0\ \chi\ \mathfrak{d}_1$. Here $\mathfrak{d}_0\ \overline{\chi^\natural} \circ \overline{\rho}\ \mathfrak{d}_1$ follows from $\Theta^e(\chi,\rho)$ using the induction hypothesis.

(Case.1): $\mathfrak{d}_j = \lhd u_j$ where $u_0\ \chi^\natural\ u_1$. Here $\mathfrak{d}_0\ \overline{\chi^\natural}\ \mathfrak{d}_1$.

(Case.2): $\mathfrak{d}_j = \mathfrak{d}\{\ldots\mathfrak{d}_{j,i}\ldots\}$ where $\mathfrak{d}_{0,i}\ \chi^\natural\ \mathfrak{d}_{1,i}$. We consider cases on the construction of the dtree context $\mathfrak{d}$ (assuming $\mathfrak{d}$ not a hole). The cases $\mathfrak{d}_j = \mathsf{Mt}$, $\mathfrak{d}_j = \lhd u_j$, and $\mathfrak{d}_j = \lambda\mathfrak{d}_{a,j}$ are trivial.

In the case $\mathfrak{d}_j = \mathsf{app}(\mathfrak{d}_{a,j},\mathfrak{d}_{b,j})$ we have $\mathfrak{d}_{a,0}\ \chi^\natural\ \mathfrak{d}_{a,1}$ and $\mathfrak{d}_{b,0}\ \chi^\natural\ \mathfrak{d}_{b,1}$. By definition of evaluation and the induction hypothesis there are $u_{a,i}$ and $u_{b,i}$ for $i = 0,1,2$ such that

$$\mathfrak{d}_{a,0} \hookrightarrow u_{a,0} \quad\text{and}\quad \mathfrak{d}_{b,0} \hookrightarrow u_{b,0} \quad\text{and}\quad u_{a,0}{}'u_{b,0} \hookrightarrow v_0,$$

$$\mathfrak{d}_{a,1} \hookrightarrow u_{a,1} \quad\text{and}\quad \mathfrak{d}_{b,1} \hookrightarrow u_{b,1}.$$

$$u_{a,0}\ \chi^\natural\ u_{a,2}\ \rho\ u_{a,1} \quad\text{and}\quad u_{b,0}\ \chi^\natural\ u_{b,2}\ \rho\ u_{b,1},$$

Since the application is defined we know that $u_{a,0} \in \mathbb{O} \wedge u_{b,0} \in \mathbb{D}^*$ or $u_{a,0} \in \mathbb{P}$. The case $u_{a,0} \in \mathbb{O}$ is trivial. If $u_{a,0} \in \mathbb{P}$ then by the definition of $u_{a,0}\ \chi^{\natural}\ u_{a,2}$ there are two non-trivial cases: (0) $u_{a,0}\ \chi\ u_{a,2}$ and (2) $u_{a,j} = u_a\{\ldots\mathfrak{d}_{j,i}\ldots\}$ with $\mathfrak{d}_{0,i}\ \chi^{\natural}\ \mathfrak{d}_{1,i}$. In the case (0), by Θ^a, $(u_{a,0}{}'u_{b,0})\ \chi^{\natural} \circ \rho\ (u_{a,2}{}'u_{b,2})$. In case (2), by definition of $\chi^{\natural}$, $(u_{a,0}{}'u_{b,0})\ \chi^{\natural}\ (u_{a,2}{}'u_{b,2})$. In either case by the induction hypothesis and $\rho \in C$ we have $(u_{a,0}{}'u_{b,0})\ \overline{\chi^{\natural}} \circ \overline{\rho}\ (u_{a,1}{}'u_{b,1})$. The if-, cart-, fst-, and rst- cases are similar. $\square_{claim.e}$

VI.4.3. Corollaries of the dtree extension theorem

The following corollaries of the dtree extension theorem are the basis for the examples to be constructed. The first corollary shows how the dtree extension condition provides criteria for constructing invertible comparisons.

▪ **Corollary 1. Invertible extension.** If ρ is an invertible comparison and the dtree extension condition is satisfied by both (χ, ρ) and (χ^-, ρ^-) then the dtree extension of ρ by χ is an invertible comparison.

$$\rho \in C_i \wedge \Theta(\chi, \rho) \wedge \Theta(\chi^-, \rho^-) \rightarrow \rho\natural\chi \in C_i$$

Proof: This follows from the definition of invertible comparison, two applications of the dtree extension theorem and the fact that $(\rho\natural\chi)^- = (\rho^-\natural\chi^-)$. □

▪ **Corollary 2. Evaluation extensions.** If ρ is a comparison and χ is contained in the evaluation closure of ρ then the dtree extension of ρ by χ is a comparison.

$$\rho \in C_i \wedge \chi \subset \overline{\rho} \rightarrow \rho\natural\chi \in C$$

Proof: By the dtree extension theorem we need only verify $\Theta(\chi, \rho)$. $\Theta^e(\chi, \rho)$ follows directly from $\chi \subset \overline{\rho}$. To verify $\Theta^a(\chi, \rho)$ note that by properties of evaluation closure $\varphi_0\ \chi\ \varphi_1$ implies $\varphi_0\ \rho\ \varphi_1$. Thus by the definition $\chi^{\natural}$ and $\rho \in C$ it follows that $\varphi_0\ \chi\ \varphi_1$ and $u_0\ \chi^{\natural}\ u_1$ implies $(\varphi_0{}'u_0)\ \chi^{\natural}\ (\varphi_0{}'u_1)\ \rho\ (\varphi_1{}'u_1)$. □

▪ **Corollary 3. Invertible evaluation extension.** If ρ is an invertible comparison and χ is contained in the invertible evaluation closure of ρ then the dtree extension of ρ by χ is an invertible comparison.

$$\rho \in C_i \wedge \chi \subset \rho \rightarrow \rho\natural\chi \in C$$

Proof: This follows from corollary 1,2 and $\overline{\rho}^{i} = \overline{\rho} \cap \left(\overline{\rho^-}\right)^-$. □

■ **A substitution closure operation.** If ρ is a comparison then $\rho\natural\rho$ is the least substitution-closed comparison containing ρ. If ρ is an invertible comparison then $\rho\natural\rho$ is the least invertible substitution-closed comparison containing ρ.

$$Cl_{dsubst}(\rho\natural\rho)$$

$$\rho \in \mathcal{C} \rightarrow (\rho\natural\rho \in \mathcal{C}) \wedge (\rho \subset \rho_0 \in \mathcal{C} \wedge Cl_{dsubst}(\rho_0) \rightarrow \rho\natural\rho \subset \rho_0)$$

$$\rho \in \mathcal{C}_{\mathrm{i}} \rightarrow (\rho\natural\rho \in \mathcal{C}_{\mathrm{i}}) \wedge \rho \subset \rho_0 \in \mathcal{C}_{\mathrm{i}} \wedge Cl_{dsubst}(\rho_0) \rightarrow \rho\natural\rho \subset \rho_0$$

Proof: This follows from corollaries 2 and 3, the dtree extension lemma, and preservation laws for substitution closure using the fact that for comparisons $\rho \subset \overline{\rho}^{i} \subset \overline{\rho}$.

VI.4.4. Example comparison constructions

We will define a number of comparisons using the dtree extension operation. We call these comparisons "transforms" as they can be thought of as rules for transforming programs.

Partial Evaluation transforms.

If $\partial \hookrightarrow v$ then dtrees related by replacing some occurrences of ∂ by value nodes with value v generate a comparison relation extending dtree equality that corresponds to partial evaluation. Partial evaluation transforms are tree pruning relations. The computation trees described by related dtrees are related by replacing some subtrees by references to their value. An example is replacing occurrences of Add1(Add1(0)) by 2. Any set of evaluations generates a comparison, for example, evaluation of all dtrees that correspond to terms built only from data and data operations. The complete partial evaluation transformation is obtained by using the set of all evaluations.

▹ **Complete partial evaluation transform.**

$$\overset{ev}{\mapsto} \underset{\mathrm{df}}{\equiv} \rho_{=}\natural\chi_{\mathrm{ev}} \quad \text{where} \quad \chi_{\mathrm{ev}} = \{(\partial, \triangleleft v) \mid \partial \hookrightarrow v\}$$

■ Since $\rho_{=} \in \mathcal{C}_{\mathrm{i}}$ and $\chi_{\mathrm{ev}} \subset \overline{\rho_{=}}^{i}$ we have by Corollary 3 that $\overset{ev}{\mapsto} \in \mathcal{C}_{\mathrm{i}}$.

Syntactic transforms

Many comparison relations are naturally expressed in terms of relations on forms. We call these relations *syntactic transforms.*

▷ Let X be a set of pairs of forms. We define the dtree extension of ρ by X (written $\overset{\rho,X}{\mapsto}$) as follows.

$$\overset{\rho,X}{\mapsto} \underset{\mathrm{df}}{=} \rho\natural\chi_{\mathrm{X}} \quad \text{where} \quad \chi_{\mathrm{X}} = \{(\langle\mathfrak{f}_0 \mid \xi\rangle, \langle\mathfrak{f}_1 \mid \xi\rangle) \mid (\mathfrak{f}_0, \mathfrak{f}_1) \in \mathrm{X}, \xi \in \mathbb{E}\}$$

■ Expressing Corollaries 2 and 3 in terms of forms we have

$$\rho \in \mathcal{C} \land X \subset \overline{\rho} \to \overset{\rho,X}{\mapsto} \in \mathcal{C} \quad \text{and} \quad \rho \in \mathcal{C}_{\mathrm{i}} \land X \subset \overline{\rho}^{\,i} \to \overset{\rho,X}{\mapsto} \in \mathcal{C}_{\mathrm{i}}$$

As examples we define

$$X_{if.if} \underset{\mathrm{df}}{=} \{\mathrm{if}(\mathrm{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2),\mathfrak{f}_3,\mathfrak{f}_4),\ \mathrm{if}(\mathfrak{f}_0,\mathrm{if}(\mathfrak{f}_1,\mathfrak{f}_3,\mathfrak{f}_4),\mathrm{if}(\mathfrak{f}_2,\mathfrak{f}_3,\mathfrak{f}_4))\}$$

$$X_{if.app} \underset{\mathrm{df}}{=} \{(\mathfrak{f}(\mathrm{if}(\mathfrak{f}_0,\mathfrak{f}_1,\mathfrak{f}_2)),\ \mathrm{if}(\mathfrak{f}_0,\mathfrak{f}(\mathfrak{f}_1),\mathfrak{f}(\mathfrak{f}_2)))\}$$

$$X_{if.or} \underset{\mathrm{df}}{=} \{(\mathrm{if}(\mathrm{or}(\mathfrak{f}_0,\mathfrak{f}_1),\mathfrak{f}_2,\mathfrak{f}_3),\ \mathrm{if}(\mathfrak{f}_0,\mathfrak{f}_2,\mathrm{if}(\mathfrak{f}_1,\mathfrak{f}_2,\mathfrak{f}_3)))\}$$

$$X_{let.perm} \underset{\mathrm{df}}{=} \{(\mathrm{let}\{s_0 \leftarrow \mathrm{let}\{s_1 \leftarrow \mathfrak{f}_1\}\mathfrak{f}_0\}\mathfrak{f},\ \mathrm{let}\{s_1 \leftarrow \mathfrak{f}_1\}\mathrm{let}\{s_0 \leftarrow \mathfrak{f}_0\}\mathfrak{f}) \\ \mid s_1 \notin \mathit{frees}(\mathfrak{f})\}$$

$$X_{cart.id} \underset{\mathrm{df}}{=} \{(\mathrm{cart}(\mathfrak{f},\mathrm{mt}),\ \mathfrak{f})\}$$

$$X_{cart.assoc} \underset{\mathrm{df}}{=} \{(\mathrm{cart}(\mathrm{cart}(\mathfrak{f}_0,\mathfrak{f}_1),\mathfrak{f}_2),\ \mathrm{cart}(\mathfrak{f}_0,\mathrm{cart}(\mathfrak{f}_1,\mathfrak{f}_2)))\}$$

$$X_{let.elim} \underset{\mathrm{df}}{=} \{(\mathrm{let}\{s \leftarrow \mathfrak{f}_0\}\mathfrak{f},\ \mathfrak{f}|^{s}_{\mathfrak{f}_0})\}$$

$$X_{let.intro} \underset{\mathrm{df}}{=} \{(\mathfrak{f}|^{s}_{\mathfrak{f}_0},\ \mathrm{let}\{s \leftarrow \mathfrak{f}_0\}\mathfrak{f}) \mid \Downarrow(\mathfrak{f}|^{s}_{\mathfrak{f}_0}) \to \Downarrow\mathfrak{f}_0\}.$$

Then taking ρ to be $\rho_{=}$ (or any invertible comparison) and X to be one of the sets $X_{if.dist}$, $X_{if.or}$, $X_{let.perm}$, $X_{cart.id}$, or $X_{cart.assoc}$ we have $X \subset \overline{\rho}^{\,i}$ and $\overset{\rho,X}{\mapsto} \in \mathcal{C}_{\mathrm{i}}$. Furthermore by the preservation properties given in §1 the symmetric closure $\uplus\{\overset{\rho,X}{\mapsto}, (\overset{\rho,X}{\mapsto})^{-}\}$ of $\overset{\rho,X}{\mapsto}$ is a symmetric comparison. Thus any invertible comparison can be extended to a symmetric comparison satisfying the (if.if), (if.app), (if.or), (let.perm) (cart.id), and (cart.assoc) laws for $\cong$ given in §2. In particular, this verifies these laws for $\cong$.

If ρ is any comparison containing $(\overset{ev}{\mapsto})^{-}$ and $X = X_{let.elim}$ then $\overset{\rho,X}{\mapsto}$ is a comparison closed under the let-elimination transform. If ρ is any invertible comparison containing $\overset{ev}{\mapsto}$ and $X = X_{let.intro}$ then $\overset{\rho,X}{\mapsto}$ is an invertible comparison closed under let-introduction. Further, if $X = X_{let.intro}$ then $\uplus\{\overset{\rho,X}{\mapsto}, (\overset{\rho,X}{\mapsto})^{-}\}$ satisfies the (let.cnv) law given in §2 and thus we verify this law for $\cong$.

Abstraction transforms

As a final example we define the one step abstraction closure of a comparison.

▷ **One step abstraction closure.** The one step abstraction closure $\overset{\rho,abs}{\mapsto}$ of ρ is defined by

$$\overset{\rho,abs}{\mapsto} \underset{\text{df}}{=} \rho^{\natural}\chi_{\mathbf{abs}} \quad \text{where} \quad \chi_{\mathbf{abs}} = \{(\varphi_0, \varphi_1) \mid (\forall v)((\varphi_0 {}' v)\ \rho\ (\varphi_1 {}' v))\}$$

■ If ρ is a comparison then the one step abstraction closure of ρ is a comparison.

$$\rho \in \mathcal{C} \rightarrow \overset{\rho,abs}{\mapsto} \in \mathcal{C}.$$

Proof: By the dtree extension theorem we need only verify $\Theta(\chi_{\mathbf{abs}}, \rho)$. For $\Theta^e(\chi_{\mathbf{abs}}, \rho)$ note that since $\chi_{\mathbf{abs}}$ is a pfn relation we have $\mathfrak{d}_0\ \chi_{\mathbf{abs}}\ \mathfrak{d}_1$ implies $\mathfrak{d}_0\ \overline{\chi_{\mathbf{abs}}}\ \mathfrak{d}_1$. For $\Theta^a(\chi_{\mathbf{abs}}, \rho)$ note that $\varphi_0\ \chi_{\mathbf{abs}}\ \varphi_1$ and $u_0\ \chi^{\natural}_{\mathbf{abs}}\ u_1$ implies $(\varphi_0 {}' u_0)\ \chi^{\natural}_{\mathbf{abs}}\ (\varphi_0 {}' u_1)\ \rho\ (\varphi_1 {}' u_1)$. □

■ The one step abstraction closure preserves invertibility and symmetry.

$$\rho \in \mathcal{C}_{\mathrm{i}} \rightarrow \overset{\rho,abs}{\mapsto} \in \mathcal{C}_{\mathrm{i}} \quad \text{and} \quad \rho = \rho^{-} \rightarrow \overset{\rho,abs}{\mapsto} = (\overset{\rho,abs}{\mapsto})^{-}$$

Remarks.

- Notice that taking ρ to be $\sqsubseteq$ we have an alternative proof of the extensionality theorem.

- The one step abstraction closure operation is a monotone operation on comparisons. Thus it can be iterated along the ordinals to a fixed point which will be the least extensional comparison containing the initial comparison. (This is a form of inductive definition – see for example Moschovakis [1975]). This is an alternative to the construction of the least extensional comparison by intersecting all of the extensional comparisons containing the given starting relation.

Chapter VII. Abstract machines and compiling morphisms

Now we return to the world of sequential computation. In this chapter we generalize the idea of introducing a continuation parameter in order to transform recursive pfns to sequential pfns. This idea was used in the tree product example (§II.2) to transform a recursive computation into a sequential one. We will show that

(†) ***any* s-*Rum* *computation can be described by a sequential dtree of* t-*Rum* *and carried out using the reduces-to relation.***

Notions of *Rum* machine structure and machine morphism are introduced informally in order to provide mechanisms for relating different representations of computation descriptions and to formulate precisely the claim (†). There is a natural *Rum* machine structure $\mathcal{R}$ on s-*Rum*. A *Rum* machine structure $\mathcal{T}$ is defined on t-*Rum* and a map from $\mathcal{R}$ to $\mathcal{T}$ is defined and proved to be a morphism. We think of $\mathcal{T}$ as an abstract machine and the morphism as a naive compiler. A simple example, compiling and optimizing the pfn Car, is given to illustrate how program transformations can be combined with naive compiling to produce plausible results.

VII.1. *Rum* machine structures

A *Rum* machine structure over $(\mathfrak{D},\mathbb{S}_y)$ has a collection of states and a step relation on these states. In addition, there are forms, environments and dtrees for describing computations and continuations for describing computation contexts. The computation domain contains representations of data, data operations, pfns and continuations. There are injections $(\iota_s, \iota_d, \iota_o)$ mapping $\mathbb{S}_y$ to machine forms, and mapping $\mathbb{D}$ and $\mathbb{O}$ into the computation domain. Environments are finite maps from symbols to sequences from the computation domain. There are construction operations including a closure operation (*close*) mapping form - environment pairs to dtrees; an application operation (*appl*) mapping data operation - data sequence pairs and continuation - value pairs to dtrees; an abstraction operation (*pfn*) mapping symbol - form - environment triples into the computation domain (the representation of pfns); an injection (*note*) from continuations into the computation domain (the representation of continuations); an identity continuation (*idcont*), a begin operation (*begin*) mapping continuation - dtree pairs to states; and a return operation (*return*) mapping continuation - value pairs to states. Finally there is a next state operation (*step*). The domains and operations of a *Rum* machine structure are summarized in Figure 19 using generic *Rum* notation for domains.

Domains

$$<\mathbb{F}, \mathbb{E}, \mathbb{V}, \mathbb{D}\mathrm{t}, \mathbb{C}\mathrm{o}, \mathbb{S}\mathrm{t}>$$

Embedding of ($\mathfrak{D}$, $\mathbb{S}\mathrm{y}$)

$\iota_s : [\mathbb{S}\mathrm{y} \to \mathbb{F}]$ % an injection

$\iota_d : [\mathbb{D} \to \mathbb{V}]$ % an injection

$\iota_o : [\mathbb{O} \to \mathbb{V}]$ % an injection

$\mathbb{E} = [\iota_s(\mathbb{S}\mathrm{y}) \ast\!\succ \mathbb{V}^*]$

Machine Operations

$close : [\mathbb{F} \times \mathbb{E} \to \mathbb{D}\mathrm{t}]$

$appl : [(\mathbb{O} \times \mathbb{D}^*) \oplus (\mathbb{C}\mathrm{o} \times \mathbb{V}^*) \to \mathbb{D}\mathrm{t}]$

$pfn : [\mathbb{S}\mathrm{y} \times \mathbb{F} \times \mathbb{E} \to \mathbb{V}]$

$note : [\mathbb{C}\mathrm{o} \to \mathbb{V}]$ % an injection

$idcont : [\square \to \mathbb{C}\mathrm{o}]$

$begin : [\mathbb{C}\mathrm{o} \times \mathbb{D}\mathrm{t} \to \mathbb{S}\mathrm{t}]$

$return : [\mathbb{C}\mathrm{o} \times \mathbb{V}^* \to \mathbb{S}\mathrm{t}]$

$step : [\mathbb{S}\mathrm{t} \rightsquigarrow \mathbb{S}\mathrm{t}]$

Figure 19. *Rum* machine structure over ($\mathfrak{D}$,$\mathbb{S}\mathrm{y}$)

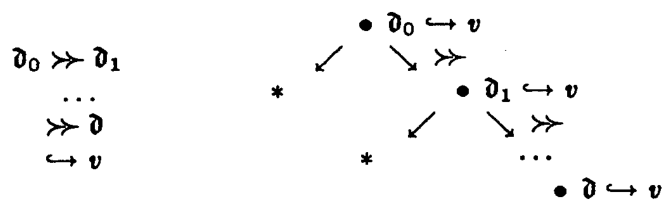
The s-*Rum* world described in Chapter V has a natural machine structure, $\mathcal{R}$. So far we have imagined that the injection mapping symbols to forms and the injections mapping data, data operations and continuations into the computation domain are simply inclusions. When necessary to distinguish objects from their representations in the computation domain we write $d \in \mathbb{V}$, $o \in \mathbb{V}$, and $\gamma \in \mathbb{V}$. $pfn(s, \mathfrak{f}, \xi) = \langle\lambda(s)\mathfrak{f} \mid \xi\rangle$ which we write as $\langle\lambda(s)\mathfrak{f} \mid \xi\rangle \in \mathbb{V}$ if we need to distinguish the pfn from the dtree of the same structure.

In order to talk about more than one machine structure we relativize the notation for a *Rum* world by adding a superscript denoting the machine. To avoid notational clutter, superscripts will generally be omitted from operation and relation symbols since the interpretation can be determined by the arguments. For objects of $\mathcal{R}$ we omit superscripts (this is the default interpretation) except for emphasis. For example, $\mathfrak{f}$ (or $\mathfrak{f}^{\mathcal{R}}$) is a form of $\mathcal{R}$, and we write $\mathbb{F}^{\mathcal{R}}$ or just $\mathbb{F}$ for the set of forms of $\mathcal{R}$. More generally, if $\mathcal{A}$ is a *Rum* machine structure, $\mathfrak{f}^{\mathcal{A}}$ is a form of $\mathcal{A}$ and $\mathbb{F}^{\mathcal{A}}$ is the set of forms (the interpretation of $\mathbb{F}$) in $\mathcal{A}$. The image of s in $\mathbb{F}^{\mathcal{A}}$ will be denoted by $s^{\mathcal{A}}$ and the images of d and o in $\mathbf{V}^{\mathcal{A}}$ will be denoted by $d^{\mathcal{A}}$, $o^{\mathcal{A}}$. $\gamma \in \mathbf{V}^{\mathcal{A}}$ is the representation of $\gamma^{\mathcal{A}}$ in $\mathbf{V}^{\mathcal{A}}$. We will use s-*Rum* notation for machine operations. Thus we write $\langle \mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}} \rangle$ for $close(\mathfrak{f}^{\mathcal{A}}, \xi^{\mathcal{A}})$ and $\vartheta^{\mathcal{A}\,\prime} v^{\mathcal{A}}$ for $appl(\vartheta^{\mathcal{A}}, v^{\mathcal{A}})$. The application notation is extended to pfns as usual by

$$\langle \lambda(s^{\mathcal{A}})\mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}} \rangle\,'\, v^{\mathcal{A}} = \langle \mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}}\{s^{\mathcal{A}} \leftarrow v^{\mathcal{A}}\} \rangle.$$

VII.2. The tree machine $\mathcal{T}$

A tree machine is a machine structure defined on t-*Rum*. Tree machine states are *sequential* dtrees – dtrees that can be evaluated using reduces-to steps with only limited side computations. The computation trees described by sequential dtrees look like lists with reductions along the main (rightmost) branch and side computations as elements of the list. This is illustrated in Figure 20.


$\partial_0 \twoheadrightarrow \partial_1$
...
$\twoheadrightarrow \partial$
$\hookrightarrow v$
● $\partial_0 \hookrightarrow v$
$\twoheadrightarrow$
*
● $\partial_1 \hookrightarrow v$
$\twoheadrightarrow$
*
...
● $\partial \hookrightarrow v$

* stands for a side computation.

Figure 20. Sequential dtrees

For example the dtrees formed by application of Tprod1 to number trees are sequential dtrees (see §II.2). The key in describing sequential computation of the tree product function was to make the continuation of the computation an explicit parameter (a continuation pfn). In such computations application is used in several ways: to apply a data operation to a data sequence, to return a value to a continuation, and to call a pfn with an argument and a continuation.

Generalizing the basic ideas involved in the transformation of the tree product computation we define the tree machine structure, $\mathcal{T}$. Notation, rules for generating $\mathcal{T}$-objects, and definitions of $\mathcal{T}$-operations and injections are given in Figure 21. In order to keep track of the different roles played by pfns in the tree machine and to distinguish the various uses of application, symbols are partitioned into three sorts: value symbols, operation symbols, and continuation symbols. There are just two continuation symbols which we denote by c and $\mathsf{c}*$. Further we fix an injection from $\mathbb{S}y$ into the value symbols, $s \mapsto s^{\mathcal{T}}$, and an isomorphism of $\mathbb{O}$ with the operation symbols, $o \leftrightarrow \ddot{o}$.

In addition to the required sorts of machine objects, $\mathcal{T}$ has *value instructions* $\mathfrak{f}^v$ and *state forms* $\mathfrak{f}^s$. Value instructions are forms describing the primitive computations that generate values: reference to a value bound in the environment (vsym), pfn formation (pfn), noting (note), application of data operations to data sequences (dapp), and instructions for sequence manipulation (mt,cart,fst,rst). State forms are generated by constructions corresponding to returning a value to a continuation (ret), conditional branching (if), calling a pfn with an argument and a continuation (call), and making temporary bindings of continuations and values (cbnd, vbnd). State-forms serve as symbolic descriptions of computation states. Each state-form contains a free continuation symbol whose intended interpretation is a pfn representing the current context.

$\mathcal{T}$-environments are environments that bind $\mathcal{T}$-values to value symbols (and that bind no other symbols). A $\mathcal{T}$-environment extended by binding the continuation symbol c to a $\mathcal{T}$-continuation is called a $\mathcal{T}$-state-environment. For forming closures in $\mathcal{T}$ we will assume we are working in a global context binding data operations to the corresponding symbols.

The remaining tree machine objects are generated naturally from state-forms by abstraction and closure. $\mathcal{T}$-forms are obtained by abstraction of $\mathcal{T}$-state-forms with respect to the continuation symbol c. A $\mathcal{T}$-dtree is either the closure of a $\mathcal{T}$-form in a $\mathcal{T}$-environment or a continuation application dtree (capp). A $\mathcal{T}$-continuation is either the identity pfn I, or a pfn with body a $\mathcal{T}$-state-form, argument symbol a value symbol and environment an $\mathcal{T}$-state-environment. A $\mathcal{T}$-state is either a terminal state $\mathsf{I}'v^{\mathcal{T}}$ or a $\mathcal{T}$-state-form closed in a $\mathcal{T}$-state-environment.

The following facts about $\mathcal{T}$-instructions and $\mathcal{T}$-states follow easily from the definitions. The key point is that $\mathcal{T}$-states are evaluated by reduces-to steps with limited side computations and, when defined, $\mathcal{T}$-states return $\mathcal{T}$-values.

$\mathcal{T}$ notation

Variables	Range	Variables	Range
s^v	value symbols	$\mathfrak{f}^v$	value instructions
$\ddot{o}$	operation symbols	$\mathfrak{f}^s$	state forms
c, $\mathsf{c*}$	continuation symbols		

Value instructions / State forms

Value instructions				State forms	
(vsym)	s^v	(mt)	mt	(ret)	$\mathsf{c}(\mathfrak{f}^v)$
(dapp)	$\ddot{o}(s^v)$	(cart)	$\mathsf{cart}(s_l^v, s_r^v)$	(if)	$\mathsf{if}(\mathfrak{f}^v, \mathfrak{f}_t^s, \mathfrak{f}_e^s)$
(abs)	$\lambda(s^v,\mathsf{c})\mathfrak{f}^s$	(fst)	$\mathsf{fst}(s^v)$	(call)	$s_f^v(s_a^v,\mathsf{c})$
(note)	$\lambda(s^v,\mathsf{c*})\mathsf{c}(s^v)$	(rst)	$\mathsf{rst}(s^v)$	(cbnd)	$\mathsf{let}\{\mathsf{c} \leftarrow \lambda(s^v)\mathfrak{f}_0^s\}\mathfrak{f}^s$
				(vbnd)	$\mathsf{let}\{s^v \leftarrow \mathfrak{f}^v\}\mathfrak{f}^s$

$\mathcal{T}$-Forms: $\lambda(\mathsf{c})\mathfrak{f}^s$

$\mathcal{T}$-Dtrees: (close) $\langle\lambda(\mathsf{c})\mathfrak{f}^s \mid \xi^{\mathcal{T}}\rangle$ (capp) $\langle\lambda(\mathsf{c*})\mathsf{c}(s^v) \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}\rangle$

$\mathcal{T}$-Continuations: (id) I (cont) $\langle\lambda(s^v)\mathfrak{f}^s \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma_{\}}^{\mathcal{T}}\rangle$

$\mathcal{T}$-States: (ter) $\mathsf{I}'v^{\mathcal{T}}$ (nter) $\langle\mathfrak{f}^s \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}\rangle$

$\mathcal{T}$-Environments $[\mathbb{S}_v{}^{\mathcal{T}} *\!\succ \mathbf{V}^{\mathcal{T}^*}]$ $\xi^{\mathcal{T}}(\ddot{o}) = \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}(\ddot{o}) = o$

$\mathcal{T}$ Injections and Operations

(data)	$d^{\mathcal{T}} = d$
(dop)	$o^{\mathcal{T}} = \lambda(s^v,\mathsf{c})\mathsf{c}(\ddot{o}(s^v))$
(close)	$\mathit{close}(\mathfrak{f}^{\mathcal{T}}, \xi^{\mathcal{T}}) = \langle\mathfrak{f}^{\mathcal{T}} \mid \xi^{\mathcal{T}}\rangle$
(appl)	$\mathit{appl}(\vartheta^{\mathcal{T}}, v^{\mathcal{T}}) = \vartheta^{\mathcal{T}}\,'\,v^{\mathcal{T}}$
(pfn)	$\mathit{pfn}(s^v, \mathfrak{f}^{\mathcal{T}}, \xi^{\mathcal{T}}) = \langle\lambda(s^v)\mathfrak{f}^{\mathcal{T}} \mid \xi^{\mathcal{T}}\rangle$
(note)	$\gamma^{\mathcal{T}} \in \mathbf{V}^{\mathcal{T}} = \langle\lambda(s^v,\mathsf{c*})\mathsf{c}(s^v) \mid \mathsf{c} \leftarrow \gamma^{\mathcal{T}}\rangle$
(idcont)	I
(begin)	$\gamma^{\mathcal{T}} \triangledown \mathfrak{d}^{\mathcal{T}} = \mathfrak{d}^{\mathcal{T}}\,'\,\gamma^{\mathcal{T}}$
(return)	$\gamma^{\mathcal{T}} \triangledown v^{\mathcal{T}} = \gamma^{\mathcal{T}}\,'\,v^{\mathcal{T}}$
(step)	$\succ\!\!\succ_{\iota}$

Figure 21. The *Rum* Tree Machine

■ **Value instruction lemma.** When defined, a value instruction closed in a $\mathcal{T}$-state-environment returns a $\mathcal{T}$-value.

$$\langle \mathfrak{f}^{v} \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}\rangle \hookrightarrow v \rightarrow v \in (\mathbf{V}^{\mathcal{T}})^{*}$$

The evaluation of a value instruction in a $\mathcal{T}$-state-environment is of bounded (small) size independent of the environment. A value instruction is undefined only in the case of data operation application where the argument is not a data sequence.

■ $\mathcal{T}$-states are closed under reduces-to.

$$\mathfrak{d}_0 \in \mathbb{S}t^{\mathcal{T}} \wedge \mathfrak{d}_0 \ggcurly \mathfrak{d}_1 \rightarrow \mathfrak{d}_1 \in \mathbb{S}t^{\mathcal{T}}$$

■ If a $\mathcal{T}$-state is defined then the value is a $\mathcal{T}$-value and the state reduces to the terminal state returning that value.

$$\Downarrow\langle \mathfrak{f}^{s} \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}\rangle \rightarrow (\exists v^{\mathcal{T}})(\langle \mathfrak{f}^{s} \mid \xi^{\mathcal{T}}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}\}\rangle \ggcurly \mathsf{l}' v^{\mathcal{T}})$$

Remark. We have arranged the definitions so that each state-form has a unique free continuation symbol c. The additional continuation symbol c∗ is used only in the note instruction where it necessary to name both the continuation argument and the continuation contained in the noted environment. This is consistent with the single current continuation used in s-*Rum* computation and, using the note instruction multiple contexts can be kept in the environment. In practice continuations might be represented as actual host machine state, while the representation in the computation domain is a data structure with sufficient information to create the corresponding host machine state. Packaging and unpackaging these contexts could be expensive operations. Thus having several host machine contexts allows one to switch contexts more efficiently. It is easy to modify the definitions so that any number of free continuation symbols may occur.

VII.3. Machine morphisms

A *Rum* machine morphism (known herein as morphism) maps one machine structure $\mathcal{A}$ to another $\mathcal{B}$. The purpose of such a morphism is to provide a representation of the computation descriptions and structures of $\mathcal{A}$ as objects of $\mathcal{B}$ and to prescribe how computations of $\mathcal{A}$ are carried out in $\mathcal{B}$. As the terminology suggests, a morphism should preserve the essential features of machine structure. In particular a morphism is a family of maps – one for each domain – such that the mapping commutes with the injection mappings, the construction operations and with sequence and environment constructions. It also must commute with stepping.[1]

Let $\mathcal{A}$ and $\mathcal{B}$ be machine structures over $(\mathfrak{D}, \mathbb{S}y)$ and let $\dagger$ be a family of maps from objects of $\mathcal{A}$ to objects of $\mathcal{B}$ of the corresponding sort. We write $\mathfrak{f}^{\mathcal{A}\dagger}$ for the image of $\mathfrak{f}^{\mathcal{A}}$ under $\dagger$ letting the object sort determine which element of the family of maps $\dagger$ is being applied. The requirements for a morphism listed above are spelled out by the following items.

- $\dagger$ commutes with injection mappings

(sym) $(s^{\mathcal{A}})^\dagger = s^{\mathcal{B}}$

(data) $(d^{\mathcal{A}})^\dagger = d^{\mathcal{B}}$

(dop) $(o^{\mathcal{A}})^\dagger = o^{\mathcal{B}}$

- $\dagger$ acts componentwise on values and environments.

$$\xi^{\mathcal{A}\dagger}(s^{\mathcal{B}}) = (\xi^{\mathcal{A}}(s^{\mathcal{A}}))^\dagger \quad \text{and} \quad [a_1^{\mathcal{A}} \ldots a_n^{\mathcal{A}}]^\dagger = [a_1^{\mathcal{A}\dagger} \ldots a_n^{\mathcal{A}\dagger}]$$

- $\dagger$ commutes with construction operations

(close) $\langle \mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}} \rangle^\dagger = \langle \mathfrak{f}^{\mathcal{A}\dagger} \mid \xi^{\mathcal{A}\dagger} \rangle$

(appl) $(o^{\mathcal{A}}{}' d^{\mathcal{A}})^\dagger = (o^{\mathcal{A}\dagger})'(d^{\mathcal{A}\dagger})$

$(\gamma^{\mathcal{A}}{}' v^{\mathcal{A}})^\dagger = (\gamma^{\mathcal{A}\dagger})'(v^{\mathcal{A}\dagger})$

(pfn) $\langle \lambda(s^{\mathcal{A}})\mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}} \rangle^\dagger = \langle \lambda(s^{\mathcal{A}\dagger})\mathfrak{f}^{\mathcal{A}\dagger} \mid \xi^{\mathcal{A}\dagger} \rangle$

(note) $(\gamma^{\mathcal{A}} \in \mathbf{V}^{\mathcal{A}})^\dagger = (\gamma^{\mathcal{A}\dagger}) \in \mathbf{V}^{\mathcal{B}}$

(idcont) $\mathsf{Id}^{\mathcal{A}\dagger} = \mathsf{Id}^{\mathcal{B}}$

(begin) $(\gamma^{\mathcal{A}} \triangledown \boldsymbol{\partial}^{\mathcal{A}})^\dagger = (\gamma^{\mathcal{A}\dagger}) \triangledown (\boldsymbol{\partial}^{\mathcal{A}\dagger})$

(return) $(\gamma^{\mathcal{A}} \vartriangle v^{\mathcal{A}})^\dagger = (\gamma^{\mathcal{A}\dagger}) \vartriangle (v^{\mathcal{A}\dagger})$

[1] We consider here only morphisms that commute with stepping in a very strong sense. This requirement is adequate for our present purpose, but is too limiting in general. What is needed is the existence of "derived" stepping relations such that the morphism carries a derived relation for machine $\mathcal{A}$ to a derived relation for machine $\mathcal{B}$.

- † commutes with stepping: if $\varsigma_0^{\mathcal{A}} \rightarrowtail_\iota \varsigma_0^{\mathcal{A}}$ then $\varsigma_0^{\mathcal{A}\dagger} \rightarrowtail_\iota \varsigma_1^{\mathcal{A}\dagger}$ and if $\varsigma_0^{\mathcal{B}} = \varsigma_0^{\mathcal{A}\dagger}$ and $\varsigma_0^{\mathcal{B}} \rightarrowtail_\iota \varsigma_1^{\mathcal{B}}$ then $\varsigma_0^{\mathcal{A}} \rightarrowtail_\iota \varsigma_1^{\mathcal{A}}$ where $\varsigma_1^{\mathcal{B}} = \varsigma_1^{\mathcal{A}\dagger}$. This is summarized in the diagram below.

$$\begin{tikzcd}
\varsigma_0^{\mathcal{A}} \arrow[r, tail, "\iota"'] \arrow[d, "\dagger"] & \varsigma_1^{\mathcal{A}} \arrow[d, "\dagger"] \\
\varsigma_0^{\mathcal{B}} \arrow[r, tail, "\iota"'] & \varsigma_1^{\mathcal{B}}
\end{tikzcd}$$

■ A simple consequence of the morphism requirements is that † also commutes with application of pfns to values.

$$(\langle\lambda(s^{\mathcal{A}})\mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}}\rangle\,'\,v^{\mathcal{A}})^\dagger = (\langle\lambda(s^{\mathcal{A}})\mathfrak{f}^{\mathcal{A}} \mid \xi^{\mathcal{A}}\rangle^\dagger)'(v^{\mathcal{A}\dagger})$$

VII.4. A morphism from $\mathcal{R}$ to $\mathcal{T}$

We say a $\mathcal{R}$*um* machine structure $\mathcal{A}$ is a $\mathcal{R}$*um* machine if the objects of $\mathcal{R}$ can be represented as objects of $\mathcal{A}$ and the computations of $\mathcal{R}$ can be carried out in $\mathcal{A}$. In other words, $\mathcal{A}$ is a $\mathcal{R}$*um* machine if there is a machine morphism from $\mathcal{R}$ to $\mathcal{A}$. To show that $\mathcal{T}$ is a $\mathcal{R}$*um* machine we define a morphism † from $\mathcal{R}$ to $\mathcal{T}$. From the requirements for morphisms we see that a morphism from $\mathcal{R}$ is determined by its action on forms and continuations. The basis of the definition of the map † is a map ‡ from forms to state-forms. † on forms is defined by

$$\mathfrak{f}^\dagger = \lambda(c)\mathfrak{f}^\ddagger.$$

The action of † on continuations is essentially determined by ‡ and the strong commuting requirement for stepping. The maps ‡ on forms and and † on continuations are defined in Figure 22.

A form with an immediate value maps to $\mathsf{c}(\mathfrak{f}^v)$, where $\mathfrak{f}^v$ is a value instruction generating the corresponding value. [Recall that c is the current continuation symbol.] For example, $\mathsf{mt}^\ddagger = \mathsf{c}(\mathsf{mt})$ and we have

$$\begin{tikzcd}
\gamma \triangledown \mathsf{mt} \arrow[r, tail, "\iota"'] \arrow[d, "\dagger"] & \gamma \vartriangle \square \arrow[d, "\dagger"] \\
\langle \mathsf{c}(\mathsf{mt}) \mid \mathsf{c} \leftarrow \gamma^\dagger \rangle \arrow[r, twoheadrightarrow, "\iota"'] & \gamma^{\dagger}{}'\square
\end{tikzcd}$$

The image of a form describing a composite computation is $\mathsf{let}\{\mathsf{c} \leftarrow \lambda(s^v)\mathfrak{f}_1^s\}\mathfrak{f}_0^s$ where $\mathfrak{f}_0^s$ is the image of the form describing the first subcomputation and $\lambda(s^v)\mathfrak{f}_1^s$

Mapping forms to state-forms

$$s^{\ddagger} = \mathsf{c}(s^{\mathcal{T}}) \qquad \mathsf{mt}^{\ddagger} = \mathsf{c}(\mathsf{mt})$$

$$(\lambda(s)\mathfrak{f}_{\mathrm{body}})^{\ddagger} = \mathsf{c}(\lambda(s^{\mathcal{T}},\mathsf{c})\mathfrak{f}_{\mathrm{body}}{}^{\ddagger})$$

$$(\mathsf{if}(\mathfrak{f}_{\mathrm{test}},\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{ifi}(\mathfrak{f}_{\mathrm{then}}{}^{\ddagger},\mathfrak{f}_{\mathrm{else}}{}^{\ddagger})\}\mathfrak{f}_{\mathrm{test}}{}^{\ddagger}$$

$$(\mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{appi}(\mathfrak{f}_{\mathrm{arg}}{}^{\ddagger})\}\mathfrak{f}_{\mathrm{fun}}{}^{\ddagger}$$

$$(\mathsf{cart}(\mathfrak{f}_{\mathrm{lhs}},\mathfrak{f}_{\mathrm{rhs}}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{carti}(\mathfrak{f}_{\mathrm{rhs}}{}^{\ddagger})\}\mathfrak{f}_{\mathrm{lhs}}{}^{\ddagger}$$

$$(\mathsf{fst}(\mathfrak{f}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{fstc}()\}\mathfrak{f}^{\ddagger}$$

$$(\mathsf{rst}(\mathfrak{f}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{rstc}()\}\mathfrak{f}^{\ddagger}$$

$$(\mathsf{note}(s)\mathfrak{f})^{\ddagger} = \mathsf{let}\{s^{\mathcal{T}} \leftarrow \lambda(s^{v},\mathsf{c}*)\mathsf{c}(s^{v})\}\mathfrak{f}^{\ddagger}$$

Mapping continuations to cpfns

$$\mathsf{Id}^{\dagger} = \mathsf{I}$$

$$(\gamma \circ \langle \mathsf{Ifi}(\mathfrak{f}_{\mathrm{then}},\mathfrak{f}_{\mathrm{else}}) \mid \xi\rangle)^{\dagger} = \langle \mathsf{ifi}(\mathfrak{f}_{\mathrm{then}}{}^{\ddagger},\mathfrak{f}_{\mathrm{else}}{}^{\ddagger}) \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle$$

$$(\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle)^{\dagger} = \langle \mathsf{appi}(\mathfrak{f}_{\mathrm{arg}}{}^{\ddagger}) \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle$$

$$(\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}))^{\dagger} = \langle \mathsf{appc}(s_{\mathrm{f}}^{v}) \mid s_{\mathrm{f}}^{v} \leftarrow v_{\mathrm{fun}}{}^{\dagger},\mathsf{c} \leftarrow \gamma^{\dagger}\rangle$$

$$(\gamma \circ \langle \mathsf{Carti}(\mathfrak{f}_{\mathrm{rhs}}) \mid \xi\rangle)^{\dagger} = \langle \mathsf{carti}(\mathfrak{f}_{\mathrm{rhs}}{}^{\ddagger}) \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle$$

$$(\gamma \circ \mathsf{Cartc}(v_{\mathrm{lhs}}))^{\dagger} = \langle \mathsf{cartc}(s_{\mathrm{l}}^{v}) \mid s_{\mathrm{l}}^{v} \leftarrow v_{\mathrm{lhs}}{}^{\dagger},\mathsf{c} \leftarrow \gamma^{\dagger}\rangle$$

$$(\gamma \circ \mathsf{Fstc})^{\dagger} = \langle \mathsf{fstc}() \mid \mathsf{c} \leftarrow \gamma^{\dagger}\rangle$$

$$(\gamma \circ \mathsf{Rstc})^{\dagger} = \langle \mathsf{rstc}() \mid \mathsf{c} \leftarrow \gamma^{\dagger}\rangle$$

where

$$\mathsf{ifi}(\mathfrak{f}_1,\mathfrak{f}_2) \underset{\mathrm{df}}{=} \lambda(s_{\mathrm{test}}^{v})(s_{\mathrm{test}}^{v},\mathfrak{f}_1,\mathfrak{f}_2) \qquad \text{\% } s_{\mathrm{test}}^{v} \text{ not free in } (\mathfrak{f}_1,\mathfrak{f}_2)$$

$$\mathsf{appi}(\mathfrak{f}) \underset{\mathrm{df}}{=} \lambda(s_{\mathrm{f}}^{v})\mathsf{let}\{\mathsf{c} \leftarrow \mathsf{appc}(s_{\mathrm{f}}^{v})\}\mathfrak{f} \qquad \text{\% } s_{\mathrm{f}}^{v} \text{ not free in } \mathfrak{f}$$

$$\mathsf{carti}(\mathfrak{f}) \underset{\mathrm{df}}{=} \lambda(s_{\mathrm{l}}^{v})\mathsf{let}\{\mathsf{c} \leftarrow \mathsf{cartc}(s_{\mathrm{l}}^{v})\}\mathfrak{f} \qquad \text{\% } s_{\mathrm{l}}^{v} \text{ not free in } \mathfrak{f}$$

$$\mathsf{appc}(s_{\mathrm{f}}^{v}) \underset{\mathrm{df}}{=} \lambda(s_{\mathrm{a}}^{v})s_{\mathrm{f}}^{v}(s_{\mathrm{a}}^{v},\mathsf{c}) \qquad \mathsf{fstc}() \underset{\mathrm{df}}{=} \lambda(s^{v})\mathsf{c}(\mathsf{fst}(s^{v}))$$

$$\mathsf{cartc}(s_{\mathrm{l}}^{v}) \underset{\mathrm{df}}{=} \lambda(s_{\mathrm{r}}^{v})\mathsf{c}[s_{\mathrm{l}}^{v},s_{\mathrm{r}}^{v}] \qquad \mathsf{rstc}() \underset{\mathrm{df}}{=} \lambda(s^{v})\mathsf{c}(\mathsf{rst}(s^{v}))$$

Figure 22. Definition of $\dagger : [\mathcal{R} \to \mathcal{T}]$

is a continuation segment corresponding to the context for the begin step for the composite form. For example

$$
\begin{aligned}
&(\mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}}))^{\ddagger} = \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{appi}({\mathfrak{f}_{\mathrm{arg}}}^{\ddagger})\}{\mathfrak{f}_{\mathrm{fun}}}^{\ddagger}\\
&(\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle)^{\dagger} = \langle \mathsf{appi}({\mathfrak{f}_{\mathrm{arg}}}^{\ddagger}) \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle\\
&(\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}))^{\dagger} = \langle \mathsf{appc}(s_{\mathrm{f}}^{v}) \mid s_{\mathrm{f}}^{v} \leftarrow {v_{\mathrm{fun}}}^{\dagger}, \mathsf{c} \leftarrow \gamma^{\dagger}\rangle
\end{aligned}
$$

where appi and appc are derived form constructions defined in Figure 22. For app-states we have

$$\begin{tikzcd}
\gamma \triangledown \langle \mathsf{app}(\mathfrak{f}_{\mathrm{fun}},\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle \arrow[r, tail, "\iota"'] \arrow[d, "\dagger"] & \gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle \triangledown \langle \mathfrak{f}_{\mathrm{fun}} \mid \xi\rangle \arrow[d, "\dagger"] \\
\langle \mathsf{let}\{\mathsf{c} \leftarrow \mathsf{appi}({\mathfrak{f}_{\mathrm{arg}}}^{\ddagger})\}{\mathfrak{f}_{\mathrm{fun}}}^{\ddagger} \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle \arrow[r, two heads, "\iota"'] & \langle {\mathfrak{f}_{\mathrm{fun}}}^{\ddagger} \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}_{\mathrm{appi}}\}\rangle
\end{tikzcd}$$

where

$$\gamma^{\mathcal{T}}_{\mathrm{appi}} = \langle \mathsf{appi}({\mathfrak{f}_{\mathrm{arg}}}^{\ddagger}) \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\dagger}\}\rangle = (\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle)^{\dagger}.$$

For appi-states we have

$$\begin{tikzcd}
\gamma \circ \langle \mathsf{Appi}(\mathfrak{f}_{\mathrm{arg}}) \mid \xi\rangle \vartriangle v_{\mathrm{fun}} \arrow[r, tail, "\iota"'] \arrow[d, "\dagger"] & \gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}) \triangledown \langle \mathfrak{f}_{\mathrm{arg}} \mid \xi\rangle \arrow[d, "\dagger"] \\
\gamma^{\mathcal{T}}_{\mathrm{appi}}\,'\,{v_{\mathrm{fun}}}^{\dagger} \arrow[r, two heads, "\iota"'] & \langle {\mathfrak{f}_{\mathrm{arg}}}^{\ddagger} \mid \xi^{\dagger}\{\mathsf{c} \leftarrow \gamma^{\mathcal{T}}_{\mathrm{appc}}\}\rangle
\end{tikzcd}$$

where

$$\gamma^{\mathcal{T}}_{\mathrm{appc}} = \langle \mathsf{appc}(s_{\mathrm{f}}^{v}) \mid s_{\mathrm{f}}^{v} \leftarrow {v_{\mathrm{fun}}}^{\dagger}, \mathsf{c} \leftarrow \gamma^{\dagger}\rangle = (\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}))^{\dagger}.$$

For appc-states such that $v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}}$ is well-formed we have

$$\begin{tikzcd}
\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}) \vartriangle v_{\mathrm{arg}} \arrow[r, tail, "\iota"'] \arrow[d, "\dagger"] & \gamma \triangledown v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}} \arrow[d, "\dagger"] \\
\gamma^{\mathcal{T}}_{\mathrm{appc}}\,'\,{v_{\mathrm{arg}}}^{\dagger} \arrow[r, two heads, "\iota"'] & (v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}})^{\dagger}\,'\,\gamma^{\dagger}
\end{tikzcd}$$

If $v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}}$ is well-formed then ${v_{\mathrm{fun}}}^{\dagger}$ is a pfn, with a value argument and a continuation argument and

$$
\begin{aligned}
&(\gamma^{\mathcal{T}}_{\mathrm{appc}}\,'\,v_{\mathrm{arg}})^{\dagger} = \langle s_{\mathrm{f}}^{v}(s_{\mathrm{a}}^{v},\mathsf{c}) \mid s_{\mathrm{f}}^{v} \leftarrow {v_{\mathrm{fun}}}^{\dagger}, s_{\mathrm{a}}^{v} \leftarrow {v_{\mathrm{arg}}}^{\dagger}, \mathsf{c} \leftarrow \gamma^{\dagger}\rangle\\
&((v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}})\,'\,\gamma^{\mathcal{T}})^{\dagger} = {v_{\mathrm{fun}}}^{\dagger}\,'({v_{\mathrm{arg}}}^{\dagger}, \gamma^{\mathcal{T}\,\dagger})
\end{aligned}
$$

If $v_{\mathrm{fun}}\,'\,v_{\mathrm{arg}}$ is not well-formed then both $\gamma \circ \mathsf{Appc}(v_{\mathrm{fun}}) \vartriangle v_{\mathrm{arg}}$ and $(\gamma^{\mathcal{T}}_{\mathrm{appc}}\,'\,v_{\mathrm{arg}})^{\dagger}$ hang.

■ **Morphism theorem.** It follows from (i-iii) that † is a morphism.

(i) $\mathfrak{f}^{\ddagger}$ is a state-form for each $\mathfrak{f}$ and $\gamma^{\dagger}$ is a $\mathcal{T}$ continuation for each γ. Hence † defines a family of maps from $\mathcal{R}$ objects to $\mathcal{T}$ objects of the correct sort.

(ii) ‡ is an injection of $\mathcal{R}$-forms into $\mathcal{T}$-state-forms and † is an injection of $\mathcal{R}$ into $\mathcal{T}$.

(iii) † commutes with stepping. For the mt- and app-cases this follows from the diagrams above using injectiveness of †. The remaining cases are similar.

VII.5. Mapping and optimizing Car

Now we resume work in the global S-expression context developed in §IV.5 and apply the morphism † to the pfn Car. Define

▷ $$\mathsf{PairUnD} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x},\mathsf{c})\mathsf{c}(\mathsf{PairUn}(\mathsf{x}))$$

▷ $$\begin{aligned} \mathsf{CarD} \overset{\mathrm{df}}{\longleftrightarrow}\ & \lambda(\mathsf{x},\mathsf{c})\mathsf{let}\{\mathsf{c} \leftarrow \lambda(\mathsf{y})\mathsf{c}(\mathsf{fst}(\mathsf{y}))\} \\ & \qquad\qquad \mathsf{let}\{\mathsf{c} \leftarrow \lambda(\mathsf{f})\mathsf{let}\{\mathsf{c} \leftarrow \lambda(\mathsf{z})\mathsf{f}(\mathsf{z},\mathsf{c})\}\mathsf{c}(\mathsf{x})\} \\ & \qquad\qquad\quad \mathsf{c}(\mathsf{PairUnD}) \end{aligned}$$

Then we have

$$\mathsf{PairUn}^{\dagger} = \mathsf{PairUnD} \quad \text{and} \quad \mathsf{Car}^{\dagger} = \mathsf{CarD}$$

[D for "dagger"]. The naive "compiling" of Car makes no use of information about the global context. A natural definition of the tree machine version of Car is

▷ $$\mathsf{TmCar} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x},\mathsf{c})\mathsf{let}\{\mathsf{x} \leftarrow \mathsf{PairUn}(\mathsf{x})\}\mathsf{c}(\mathsf{fst}(\mathsf{x}))$$

and we have CarD ≅ TmCar. A more sophisticated compiler would have information about what symbols are bound to data operations in the global environment and would use the data apply instruction rather than pfn call thus producing something closer to TmCar. An alternative to modifying the compiler is to use program transformations to optimize the results of naive compiling. For example using the laws of §VI.2 we can derive TmCar from CarD as follows.

(i) $\mathsf{let\{c \leftarrow \lambda(f)let\{c \leftarrow \lambda(z)f(z,c)\}c(x)\}c(PairUnD)} \cong \mathsf{c(PairUn(x))}$

% see below

(ii) $\mathsf{let\{c \leftarrow \lambda(y)c(fst(y))\}let\{c \leftarrow \lambda(f)let\{c \leftarrow \lambda(z)f(z,c)\}c(x)\}c(PairUnD)}$

$\cong \mathsf{let\{c \leftarrow \lambda(y)c(fst(y))\}c(PairUn(x))}$

% abstraction of (i), with respect to c, note c is not global, and
% application of both sides to $\mathsf{\lambda(y)c(fst(y))}$

(iii) $\cong \mathsf{let\{y \leftarrow PairUn(x)\}c(fst(y))}$

% let-elimination in ii.rhs

(iv) $\mathsf{CarD} \cong \mathsf{TmCar}$

% abstraction of (iii) with respect to $\mathsf{(c,x)}$, using $\mathsf{c,x}$ non-global

Equation (i) is derived by a sequence of let-eliminations plus unfolding of the PairUnD definition.

$\mathsf{let\{c \leftarrow \lambda(f)let\{c \leftarrow \lambda(z)f(z,c)\}c(x)\}c(PairUnD)}$

$\cong \mathsf{let\{f \leftarrow PairUnD\}let\{c \leftarrow \lambda(z)f(z,c))\}c(x)}$ % let-elimination

$\cong \mathsf{let\{c \leftarrow \lambda(z)PairUnD(z,c)\}c(x)}$ % let-elimination

$\cong \mathsf{let\{z \leftarrow x\}PairUnD(z,c)}$ % let-elimination

$\cong \mathsf{PairUnD(x,c)}$ % let-elimination

$\cong \mathsf{\{\lambda(x,c)c(PairUn(x))\}(x,c)}$ % [unfolding PairUnD]

$\cong \mathsf{c(PairUn(x))}$ % let-elimination

Note that the let-elimination of step (iii) and most of those in the derivation of (i) are instances of

$$\mathsf{let}\{s_0 \leftarrow \lambda(s_1)\mathfrak{f}_1\}s_0(\mathfrak{f}_0) \cong \mathsf{let}\{s_1 \leftarrow \mathfrak{f}_0\}\mathfrak{f}_1$$

or in terms of application

$$\{\lambda(s_0)s_0(\mathfrak{f}_0)\}(\lambda(s_1)\mathfrak{f}_1) \cong \{\lambda(s_1)\mathfrak{f}_1\}\mathfrak{f}_0.$$

VII.6. Remarks about *Rum* machine structures and morphisms

- The combination of program transformations and machine morphisms in *Rum* provides tools for defining compilers and proving properties of compilers in a world where the computation domain contains computation abstractions such as pfns and continuations. In this context one can study and relate source to source, source to target, and target to target transformations. In practice, some of these transformations will be integrated into the compiler and some will be applied before and after compiling.

- The notions of abstract machine structure and compiling morphism can be compared to the algebraic view of structuring compilers (see for example Morris [1973] and Thatcher, Wagner, and Wright [1980]). Here there is a source language L_0, a target language L_1, a semantic domain D_0 for L_0, and a semantic domain D_1 for L_1. A compiler is a map γ from L_0 to L_1, a semantics for L_i is a map ψ_i from L_i to D_i, and an encoding is a map ϵ from D_0 to D_1. Compiler correctness with respect to the semantics and encoding maps is the requirement that the following diagram commutes

$$\begin{tikzcd} L_0 \arrow[r, "\gamma"] \arrow[d, "\psi_0"] & L_1 \arrow[d, "\psi_1"] \\ D_0 \arrow[r, "\epsilon"] & D_1 \end{tikzcd}$$

that is, $\psi_1 \circ \gamma = \epsilon \circ \psi_0$. In order to prevent trivial cases such as L_1 and D_1 being one point algebras, further requirements are needed. One possibility is to require that the encoding morphism be injective.

The initial algebra view is that L_0 is an initial $\mathcal{G}$-algebra for some signature $\mathcal{G}$. Thus the compiler γ is uniquely determined by defining a $\mathcal{G}$-algebra structure on L_1 and a semantics ψ_0 for L_0 is uniquely determined by defining a $\mathcal{G}$-algebra structure on D_0. The specification is completed by defining a $\mathcal{G}$-algebra structure on D_1 and checking that that ψ_1 and ϵ are $\mathcal{G}$-morphisms. Then $\psi_1 \circ \gamma = \epsilon \circ \psi_0$ since, by initiality, there a unique morphism from L_0 to D_1.

By definition, there is a machine morphism from $\mathcal{R}$ to each *Rum* machine $\mathcal{A}$, and this morphism is determined by its action on forms and continuations. Thus $\mathcal{R}$ is like an initial object and form and continuation constructions determine a 'signature'. Defining the image of these constructions is analogous to defining corresponding constructions in the target machine structure. The semantics that is preserved is the structure of the computations.

- The tree machine $\mathcal{T}$ and morphism † constitute a normal form theorem for *Rum* (see §I.5 for a discussion of normal form theorems). For most computation theories, a normal form theorem requires the ability to encode descriptions of computation and computation states in the underlying data structure. In *Rum* this capability is provided by the uniform abstraction and application computation primitives. This is a stronger normal form theorem than usually given in the sense that the transformation to normal form (the morphism) preserves both intensional and extensional meaning of descriptions.

- The tree machine $\mathcal{T}$ and morphism † provides a formal connection between explicit continuations (continuation pfns as an additional argument) and implicit continuations (continuations as a component of the computation state).

- Similar transformations to a sequential fragment of an AE-like language have been used by Fischer [1972] to analyze the limitations of a stack based implementation of closures and by Steele [1978] in a compiler for Scheme. Fischer essentially ignored computations that returned functions as values. Steele's treatment is very informal.

Chapter VIII. Review and concluding remarks

This chapter contains an assortment of concluding remarks. In §1 the main achievements of the work in *Rum* are reviewed. We summarize the features and what they provide as tools for reasoning about computation, and point out the main new ideas and results. In §2 we make some further remarks about our choice of basic notions. The presentation of *Rum* has been liberally sprinkled with remarks about relevant related work. However there are some additional points to be made. This is done in §3. In §4 we conclude with some discussion of applications and extensions of *Rum* and of future directions of work.

VIII.1. Review

VIII.1.1. What we can do

A variety of sorts of objects and a mixture of syntactic and semantic notions have been introduced in *Rum* in order to express naturally different aspects of computation. Forms are the basic syntactic entities. The semantic entities include environments, dtrees, pfns, computation stages, continuations, and states.

Interpretations of forms

Forms constitute a primitive programming language. The work in *Rum* provides a variety of interpretations of forms accounting for the different views of symbolic expressions that occur in practice and including several traditional types of semantics such as operational and denotational semantics, as well as an intensional semantics. In particular, we have

- Forms and semantic entities as data to be operated on – the underlying algebraic structure provides an interpretation of forms and objects of the semantic domains as data structures. The operations and relations defined (excluding comparison relations) are *Rum* computable functions on these data structures.
- Forms as descriptions of computation – viewing computation as a process of generating computation structures such as computation trees or sequences

of computation states allows us to represent properties of computations described by forms as properties of computation structures and to relate operations on descriptions to operations on the described computations. This provides an intensional semantics.

- Operational semantics – the evaluation relation provides an operational semantics – a means of determining the value denoted by a form relative to an interpretation of free symbols in the computation domain.

- Denotational semantics – the maximum equivalence $\cong$ provides a denotation of forms $\mathfrak{f}$ as a partial function $[\![\mathfrak{f}]\!]$ from environments to the computation domain modulo $\cong$.

  (denote) $$[\![\mathfrak{f}]\!](\xi) \rightsquigarrow \lfloor v \rfloor_{\cong} \leftrightarrow (\exists u \cong v)(\langle \mathfrak{f} \mid \xi \rangle \hookrightarrow u)$$

  In other words, we have an interpretation of pfns as partial functions on computation domain modulo $\cong$.

  (fun) $$\{\varphi\}_{\cong}(\lfloor v \rfloor_{\cong}) \rightsquigarrow \lfloor u \rfloor_{\cong} \underset{\text{df}}{\equiv} (\exists u_0 \cong u)(\varphi' v \hookrightarrow u_0)$$

- Towards a *Rum* calculus – the maximum equivalence contains the equations of a *Rum* calculus expressed in the language of forms. This calculus contains the laws of the call-by-value lambda calculus (Plotkin [1975]) and additional laws for if, mt, cart, fst, and rst. Using the improved recursion theorem we also have a call-by-value analog of the equational theory for LAMBDA (Scott [1976]) which contains many equations not provable in the lambda calculus.

Representation of programming tools and styles

Abstraction (pfn formation and continuation noting), application, and sequence primitives allow us to represent naturally a variety of programming styles and computation mechanisms. We have shown how to represent control and data abstraction mechanisms including systems of recursively defined functions, both first order and higher order; sequence recursion schemes; streams; tree-structured objects and tree search schemes; escape mechanisms; and co-routines. We have even illustrated how to program viewing *Rum* as a machine language.

Proving properties of programs

The recursion theorem and computation induction allow traditional first order verification methods to be represented and used in *Rum*. These include assertion methods for flowchart or loop programs (Manna [1969], [1978]); structural induction for computations over the S-expressions (Burstall [1969], Boyer-Moore[1979]); subgoal induction (Morris and Wegbreit [1976]); recursion induction (McCarthy[1963b]); and McCarthy's minimization scheme (McCarthy and Cartwright [1979]). The improved recursion theorem allows a version of Scott's fixed point induction to be represented in *Rum*.

VIII.1.2. What is new

• **The reduces-to relation.** The reduces-to relation (§IV.2) identifies subcomputations which can simply replace the parent computation rather than returning a value to it. This relates to features of the continuation-passing style (see §I.4) such as not returning a value, and computation with no buildup of external state. The reduces-to relation also plays a key role in defining machine structures on t-*Rum*.

• **Context insensitivity for sequential computation.** The dtree classification theorem (§V.3) says that even computations using context switching have useful structure. This will be an important tool in further investigations of equations satisfied by forms and dtrees in s-*Rum*.

• **Context independence for sequential computation.** The extension of computation relations of t-*Rum* such as evaluation, reduces-to, and ctree equivalence to descriptions of sequential computation (§V.3) provides tools for encapsulating context dependence and reasoning about programs as though they described tree-structured computation except when looking inside a computation that involves context noting and switching.

• **Comparison relations.** The notion of comparison relation (Chapter VI) seems not to have been formulated previously. In addition to allowing us to account for important aspects of extensional model's of the lambda calculus such as the graph model, there are some new ideas.

- using the notion of comparison relation, *extent* is derived naturally from *intent*
- the rich hierarchy of comparisons provides a variety of denotations for forms, dtrees, and pfns
- the pfn extension theorem (§VI.3) and the dtree extension theorem (§VI.4) provide tools for constructing and proving properties of comparisons.
- the notion of recursion pfn and the improved recursion theorem (§VI.2) provide a connection between the computational characterization of recursion found in computation theories and the topological, extensional characterization of recursion given by the least fixed point theorem for the graph model (see §I.5).

• **Foundations for a theory of program transformations.** The interpretation of forms as descriptions of computation together with the hierarchy of denotations for forms based on comparison relations provides both meanings to preserve and meanings to transform. Thus we have a semantic foundation for a rich theory of program transformations. In addition, transformations involving non-trivial manipulation of pfns and continuations can be treated. This was illustrated by the derivations of defining equations for Tprod1 and Tprod2 (§II.3) and by the derivation of alternative descriptions of parameterized recursion (§VI.2).

- **Machine structures and compiling morphisms.** The notions of *Rum* machine structure and morphism (Chapter VII) are a step towards developing an abstract notion of the machine underlying the *Rum* model of computation. Combined with program transformations, machines and morphisms provide a paradigm for defining and proving properties of compilers. What is new here is (i) the idea that the structure being preserved is the computation structure and (ii) in *Rum* one can express both properties preserved by program transformations and properties transformed. The representation of abstract machines in t-*Rum* (§VII.2) allows us to apply the tools developed for studying computation in t-*Rum* to problems of operating on and proving properties of machine programs. In addition, the *Rum* machine morphism defined in §VII.3 gives a formal connection between continuations as pfns and continuations as computation contexts.

- **Programming and proving with functional and control abstractions.** We have defined and proved correct several pfns describing computations not handled by traditional verification methods.

  - using continuation pfns for backtracking – the pattern matcher (Appendix B)
  - using continuations for escaping – tree product examples (§II.2) and DoUntil (§V.4)
  - using continuations for co-routining – transforming a sequence of 3-element strings to a sequence of 2-element strings (§V.5)

We have given definitions of some informal concepts such as streams (§IV.6) and co-routines (§V.5) and have begun development of the mathematical theory of these objects. Pfnls were used to describe operations on streams (§IV.6) and to describe strategies for searching tree structures (Appendix B). General properties of such pfnls were proved and used to prove properties of pfns constructed from these pfnls.

- **Derived properties and programs.** In Appendix B we define a class of properties of computation trees called derived properties and a derivation map on forms and other objects of *Rum*. We show that the derivation of a form computes the derived property of the computation tree described by the given form. In general, derivations provide a means of identifying classes of intensional properties of computations. Derivation maps provide a method with a sound semantic basis for mechanical transformation of intensional properties to extensional properties. Thus we may use the same basic tools and principles for reasoning about both aspects of programs. In particular, derivation maps can be a useful tool for extending program transformation systems such as [Scherlis 1980]. Derivations can also be used to extend mechanical theorem provers such as the Boyer-Moore theorem prover [Boyer and Moore 1979] to treat intensional as well as extensional properties of programs.

VIII.2. About the choice of basic notions

Now that we have seen something of what can be done in *Rum*, we take another look at some of choice of basic notions.

Why type-free application and abstraction as computation primitives?

Briefly, the reason is that type-free application and abstraction are simple, natural, and quite general primitives for expressing a wide variety of computation mechanisms. One disadvantage of such generality is that the fewer restrictions that are imposed the less you can say about the system. On the other hand there is no limit on the structured fragments that can be identified and studied within the general untyped framework. A further advantage is the ability to piece together substructures and diverse results smoothly, within a single framework. This is important because in the long run we want a framework suitable for mechanization and a general language for discussing computing with computers.

The work of Landin and programming examples of Burstall, Burge, Sussman and Steel, Sussman and Abelson, and Friedman et. al. demonstrates the power of AE as a programming language and language development tool. The work in semantics demonstrates the power of AE as a meta programming language capable of defining a wide variety of computation mechanisms. This provides a strong argument for AE as the basis of a theory of symbolic computation.

Why have sequences as arguments and values?

Two important reasons for taking both arguments and values to be sequences from the computation domain are simplicity and increased expressiveness. Having sequences built in as part of the basic computation primitives means that we have the simplicity of needing only a single binary application operation, while at the same time pfns are vari-ary with respect to arguments and values. This provides a natural means of expressing message-passing style computation where the argument type depends on the message type (the first element of the argument sequence). Among the examples presented in this thesis, sequences as values were used in formulating the notion of stream and in derived descriptions which return the value of derived property together with the result of the original computation. From an operational point of view, the addition of sequence primitives is like providing the programmer with primitives for accessing and manipulating the argument stack component of the computation state. In addition, the sequence primitives allow a natural formulation of bounded iteration and other sequence recursion schemes independent of the underlying data structure. Sequence primitives don't add to the recursion-theoretic power. If both sequence primitives and conditional are omitted from *Rum* we still have a call-by-value lambda calculus. In particular the partial recursive number-theoretic functions are representable in this fragment.

Why have pfns distinct from data?

More generally, why have forms, environments, dtrees, continuations, etc. distinct from data? The point is to obtain a balance of intensionality and extensionality and to study computation structures and data structures separately. To repeat the point made in §I.6, this is essential if a theory is to admit meaningful operations such as program optimizations, compiling and derivations converting intensional properties of one program into extensional properties of another. Intuitively the balance of intensionality and extensionality obtained in *Rum* is captured by the following points.

- From within a computation, pfns are extensional. A pfn can only be passed around, put into environments, or applied. Two pfns can be distinguished by another pfn only if application to some argument results in value sequences of different length or containing different corresponding data elements.
- From without, pfns have structure and computational content. They can be constructed, taken apart, and transformed. A pfn together with an argument describes how the computation of the denoted value is to be carried out.

Comparison to logical notions. The treatment of computation separately from data is similar to the formulation of first order logic in contrast to a particular first order theory such as Peano arithmetic. Computation primitives play the role of logical connectives. Data and data operations play the role of individual and function constants. An important distinction between first order logic and Peano arithmetic is that the meaning of successor is determined by its interpretation in the model of natural numbers while the meaning of logical connectives such as $\land$ or $\exists$ is determined by the notion of satisfaction which is uniformly parameterized by the interpretation of non-logical symbols in a given model.

The evaluation relation provides a semantics for the form constructions in much the same way that the notion of satisfaction provides a semantics for logical operations. Thus we might write

$$\xi \models \mathfrak{f} \hookrightarrow v$$

for

$$\langle \mathfrak{f} \mid \xi \rangle \hookrightarrow v.$$

A more general notion of satisfaction is obtained by thinking of forms as terms of a language for expressing properties of the computation domain with relation symbols denoting dtree relations. For any set of environments E, and any dtree property Φ define

$$E \models \Phi(\mathfrak{f}_0 \ldots \mathfrak{f}_n) \underset{\mathrm{df}}{\equiv} (\forall \xi \in E)\Phi(\langle \mathfrak{f}_0 \mid \xi \rangle \ldots \langle \mathfrak{f}_n \mid \xi \rangle)$$

As in a logical system, having the interpretation of expressions uniformly parameterized by the given data structure means that we have a uniform method for combining the theory of computation primitives with that of a given data structure to obtain a theory of computation over the given structure. For example, we have Peano arithmetic – a first order theory over natural numbers and $\mathcal{R}um_{\mathfrak{D}_{\mathrm{sexp}}}$ – a computation theory over S-expressions.

Why environments rather than substitution?

Forms and environments correspond to important local components of computation states. This representation of descriptions of computation allows uniform separation of syntax and semantics and of control and data. It also reflects practice. Forms and environments have independent existence as concepts in specifying, building, and extending programming systems. The separation is crucial when *Rum* is extended to a model of computation in which there are objects with internal state. It is the environment component of a pfn that has internal state. Forms remain purely static syntactic entities.

Why formulate two computation processes?

The advantage of a language describing tree-structured computations is that functional abstractions (pfns) can be viewed as functions. Rich equational theories for such languages exist and ordinary means of reasoning about partial functions can be used. The value of an expression depends only on the value of (immediate) subexpressions, and general laws for substitution hold.

The disadvantage is the complexity of the code needed to describe certain computation mechanisms which are inherently context dependent. Recall for example the difference for such simple programs as the tree product examples (Chapter II). If limited to tree-structured computation primitives, the programmer must directly represent and manage the computation state. This is not only unpleasant, but unreliable.

The introduction of control abstractions provides the best of both worlds from a programming point of view. Using the context insensitivity and context independence results for sequential computations we see that we only have to pay the price (in complexity of reasoning about programs) for control abstractions when we are really using them.

The functional interpretation of pfns describing context dependent computations can be recovered in a number of ways. The basic idea (due to Morris and Wadsworth) is to provide continuations as part of the semantic domain. To recover directly the functional interpretation, imagine that forms have an unnamed continuation parameter which is carried along during the process of computation. Each pfn is assumed to have an additional unnamed continuation argument and the current continuation is supplied along with the named argument when a pfn is applied. This is essentially the s-*Rum* interpretation. The machine morphism from the s-*Rum* to t-*Rum* provides an alternative means of recovering the functional interpretation. By the definition of morphism these two methods are isomorphic.

VIII.3. Other work

VIII.3.1. Combining and contrasting types of semantics

A key feature of *Rum* is the many views (types of semantics) provided for forms. Here we will point out other work in which alternative types of semantics are defined and used. For this purpose we focus on tree-structured computation and consider four types of semantics which we refer to as intensional, operational, equational and denotational. An intensional semantics interprets programs as descriptions of computation and provides a basis for representing properties of these computations. In *Rum* the intensional semantics is derived from the interpretation of dtrees by rules for generating computation structures. An operational semantics assigns to each program the value in the computation domain computed by the program, if any. Nothing is said about how the value is computed. In *Rum* the operational semantics is derived from the evaluation relation. An equational semantics provides axioms and rules for deducing equations between expressions of the language. This form of semantics is not yet well developed in *Rum*, but will consist of axioms and rules characterizing fragments of the maximum equivalence relation. A denotational semantics provides a semantic domain (a mathematical structure) and assigns to each program (more generally to each component expression) a denotation in the semantic domain. Denotations of composite expressions are functions of the denotations of component expressions. In *Rum* the denotational semantics of forms is obtained by interpreting the evaluation relation modulo the maximum equivalence.[1]

[1] We use the term denotational as that is the term used in the work discussed. A better term would be extensional, since the distinguishing feature of the denotations in each case is the extensionality of higher type objects in the semantic domains.

We have already discussed work relating operational semantics of AE to an equational semantics provided by a call-by-value lambda-calculus (Plotkin [1975]). Other works of relevance are Gordon [1975] treating operational and denotational semantics of Lisp; Plotkin [1977] treating operational and denotational semantics of a typed variant of AE; and Moschovakis [1984] treating intensional and denotational semantics of systems of recursive definitions.

Operational and denotational reasoning for Lisp. Gordon [1975] gives both operational and denotational semantics of pure Lisp (with dynamic binding). The two semantics are shown to be equivalent in the sense that for any Lisp expression and environment both semantics determine the same S-expression value or both determine that the value is undefined. Gordon's reason for defining the operational semantics was to provide additional tools for proving properties of programs which reflect reasoning based on operational intuitions. The equivalence of the two semantics allows one to use the semantics best suited for each given problem. The operational semantics is based on a reduction (step) relation on S-expressions coding Lisp expression - environment pairs. From this is derived a subcomputation relation and a Lisp induction principle similar to the notions of subcomputation and computation induction in *Rum*. These are the basic tools for operational reasoning. The denotational semantics provides denotations of Lisp expressions in a complete partial order using Scott-Strachey methods.

Operational and denotational equivalence in LCF. Plotkin [1977] gives operational and denotational semantics for a simple programming language based on LCF (a logic for a typed fragment of AE – see Gordon et. al. [1979]) and compares notions of operational and denotational equivalence. Programs are ground terms of LCF, which may of course involve higher type subterms. An evaluator for programs is defined and denotations are assigned in a standard LCF model (a typed family of c.p.o.s) and two extensions of this model. The operational and denotational semantics agree in the sense that they assign the same value to closed ground terms. However, natural notions of equivalence on programs derived from the two types of semantics turn out to be different. Two terms (of the same type) are denotationally equivalent if they have the same denotation in all environments. Two terms are operationally equivalent if one may be substituted for the other in any program context without changing the denotation of the program (see discussion of Plotkin [1975] in §I.4). The main results are (i) denotational equivalence implies operational equivalence, (ii) operational equivalence does not imply denotational equivalence, and (iii) the language can be extended by adding a parallel construct so that denotational and operational equivalence coincide. This provides an interesting analysis of the two notions of equivalence and illustrates the use of denotational semantics to discover additional computation primitives.

The semantic domains used for the denotations all have the property that they are generated from ground domains by taking domains of type $\sigma \rightarrow \tau$ to be the domain of all continuous functions from the domain of type σ to the domain of type τ. It is suggested that by choosing a suitably restricted set of functions for domains of type $\sigma \rightarrow \tau$ one might obtain a denotational semantics such that denotational and operational equivalence are the same for the original language. The problem of finding such domains is left open.

Foundations for a theory of algorithms. Moschovakis [1984] is a study of the notion of algorithm. The basic questions are: What sort of mathematical object is an algorithm? How are they defined and constructed?

The proposed answers are based on the idea of interpreting (descriptions of) certain systems of functionals over a given abstract structure as rules for computing the least fixed point of the system. These systems of functionals (currently called recursors) are proposed as the mathematical objects representing algorithms. The given abstract structures consist of a many sorted domain of individuals and a set of functionals acting on tuples consisting of individuals and partial functions on cartesian product spaces of individuals. The given functionals are taken as primitive recursors, i.e. as computation primitives.

A language is defined for describing algorithms. Individual terms are built from constants for given data and functionals, variables for individuals, and partial function terms by constructions including application, tupling, projections, and recursion. Partial function terms are partial function variables or terms formed by lambda abstraction from individual terms. Terms of the language are assigned both a denotation, a computable functional; and an intension, a recursor. Of course the intension viewed as a functional is the same as the denotation.

The main properties of algorithms, other than the functional computed, seem to be (i) resources used from the underlying structure and (ii) the stage of completion – a measure of the time taken to carry out the computation. In contrast to the intensional semantics provided by *Rum* the notion of algorithm does not account directly for structures generated in the process of computation or resources required to carry out the computation – such as stack. Such properties can be accounted for by transformations of control into data, i.e. by transforms, such as the compiling morphisms of *Rum*, to a subclass of programs where control resources are explicitly represented in the programs. Alternatively, by extending the given abstract structure to include data representing computation structures, one can take the algorithm described by a program to be an evaluation function together with the code for the program.

The notion of algorithm seems to capture a useful degree of intensionality. Some interesting questions to be answered are: What are useful equivalence relations for algorithms? What transformations preserve a given equivalence? Several equivalence relations have been defined and studied. The basic ideas are outlined below.[2]

- Algorithmic equivalence - two terms are algorithmically equivalent if they differ by a permutation of the variables.
- Value equivalence - two terms are value equivalent if in all structures they are assigned the same functional.
- Resource equivalence - two terms are resource equivalent if the intension of each term uses the same resources for computation in any structure.
- Resource and stage equivalence - two terms are resource and stage equivalent if the intension of each term uses the same resources for computation in any structure and computation is completed at the same stage in any structure.

Algorithmic equivalence serves the same purpose as the equivalence on form - environment pairs induced by closure dtree formation in *Rum*. Value equivalence is similar to the equivalence relation on forms obtained by requiring $\cong$-equivalence for all data structures. An interesting question is whether any of the *Rum* comparison relations capture notions of equivalence such as stage or resource equivalence.

VIII.3.2. General semantic methods

The work presented in this thesis treats a particular collection of computation primitives and structures for representing computation. The constructions of objects and definitions of relations are for the most part uniformly determined from rules corresponding the computation primitives. Thus there are some quite general principles underlying the presentation of *Rum*. In §I.5 we mentioned briefly the Scott-Strachey methods based on solving domain and semantic equations in the category of complete partial orders. Below we discuss some additional related work focusing on general methods in program semantics.

Axiomatic Operational Semantics. Plotkin [1981] proposes axiomatic presentation of operational rules for program language constructs and evaluation mechanisms as a general methodology and give examples for a variety of constructs. Functional abstractions are treated briefly, as arguments but not as values. Control abstractions are not treated.

Syntactic entities are given by a special form of inductive generation with corresponding principles of definition and proof by induction. The axioms for operational rules provide deduction systems for both static and dynamic operational

[2] These relations and results on decidability and other logical matters were presented in a series of lectures given at the CSLI workshop, Stanford, July 1985.

semantics. Static operational semantics include typing and operations on expressions such as computing the set of bound or free variables. Dynamic operational semantics provides an evaluation relation. The advantage of such a presentation by separate rules for each construct is that new constructs can be defined by adding new rules generally without modification of rules for existing constructs (assuming the new constructs can be represented using the same underlying computation structures).

The method of presentation is very similar to that of *Rum*. The difference is that Plotkin deals with operational semantics in general while *Rum* focuses on a particular system and develops many aspects of this system, including a variety of semantic notions.

Abstract Semantic Actions. Mosses [1982] proposes semantic entities called actions as the interpretation of programs. The basic ideas are illustrated by the presentation of an initial algebra generated from basic actions and operations constructing actions, together with axioms for equality of actions. Actions have several largely independent facets: they consume and produce values; they create and use bindings; they create, access and update store. The notion of facets captures in an elegant way the independence of various aspects of computation and should be important in providing useful tools for reasoning about actions.

Actions have rich expressive power, although it seems to require some work to develop intuitions about properties of actions. The example presented accounts for application and closure formation, including both static and dynamic binding. Mosses [1984] shows how mechanism such as non-local goto, labels, etc. can be treated using escape and trapping actions.

This basic approach has been used to give semantics to simple example languages and for compiler correctness. Work is in progress on a semantics for Pascal [Mosses and Watt 1985]. It will be interesting to see further development of mathematical properties of actions and how they can be used to study properties of programs and operations on programs.

Remark. Neither of the above approaches treat the intensional aspects of programs, nor are they applied to provide tools to prove properties of particular programs.

VIII.3.3. Alternative languages and models of computation

An important alternative to the *Rum* approach and to functional programming in general is the work in logic programming (see for example Lloyd [1984]). Separation of computation control and data is also an important underlying principle of logic programming. In the simplest case there is little or no control prescribed by the programmer. Programs are just axioms about the data structure which are to be used to answer queries and control is implicit in the deduction mechanism. A key tool in the process of finding answers to queries is the use of unification as a variable binding mechanism. A variety of ideas for expressing control over the search for answers have been explored. We will not attempt to discuss them here. Prolog (see Clocksin and Mellish [1984]) is the best known example of a logic programming system. More relevant to the work in *Rum* is recent work on a language called Qute (Sato and Sakauri [1984]). Qute is derived from the Lisp and Prolog traditions of computations described by symbolic expressions together with an environment. The goal of this work is to synthesize the best features of both worlds. Objects in the computation domain of Qute are called patterns. Patterns are constructed from variables and atomic constants by two different pairing operations and an abstraction operation. This is in contrast to *Rum* where free variables do not appear in values. Computations are described by expression environment pairs. The environment contains constraints on the free variables which must be satisfied.

With unification as the basic binding mechanism, Qute provides an alternative view of conditional, application, and abstraction as computation primitives. There are notions of success and failure (to unify) and conditional branches on success vs. failure. Abstraction is based on patterns rather than simple variables. Application causes the parameter pattern to be unified with the argument pattern. Thus equality $a = b$ can be defined by $\{\lambda(x, x)x\}(a, b)$.

Qute, like *Rum*, has the capability of expressing a rich variety of computations. There will be many points of interest to compare and contrast as the mathematical properties of this model of computation are developed.

VIII.4. Future directions

Rum is just the beginning, not the end of the approach presented in this thesis. It is to be extended and applied in a number of ways. Two extensions to the model of computation are: computations over memory structures (structures which have updating operations such as *rplaca* and *rplacd* of Lisp); and description of asynchronous computation à la Actors (Hewitt [1977]). Applications (putting theory into practice) include building computation systems and reasoning systems based on extensions of the *Rum* model of computation and the formalization of its metatheory. There are several directions for further research which are outlined below.

Formalization. There are two sorts of formalization to be carried out. One is formalization of *Rum* in existing formal theories (for example Feferman [1979], [1982]) whose logical complexity is well understood, in order to obtain information about the logical complexity of the maximum comparisons. The other is formalization of *Rum* and of the S-expression data structure in existing mechanical proof systems such as the Boyer-Moore theorem prover (Boyer and Moore [1979]), FOL (Weyhrauch [1980]) or EKL (Ketonen [1984]), to provide a basis for mechanical checking of proofs of properties of programs and of operations on programs. Such work can also serve as a basis for developing a program transformation checker – an interactive system for defining and carrying out program transformations.

Reflection – or – having your cake and eating it too. Excluding the comparisons, the operations and computation relations we have defined are *Rum* computable over any data structure that includes the underlying algebraic structure of *Rum*. Working over such a data structure we can extend the computation primitives to provide reflection principles which convert computation state to data and convert data to computation state. This allows programs and other objects to be treated as data within an ordinary computation. Reflection provides a mechanism for switching views within a computation – viewing an argument as data at one point and as a component of the computation state at another point. Between switches, an extensional view amenable to transformations and interpretation independent of representation is maintained, thus combining the best of both worlds. The are two ways for providing reflection. One corresponds to an isomorphic embedding of the computation theory and is the form of reflection implemented by Smith [1982]. The other corresponds to a *self-reflexive* embedding which is the form of reflection implicit in traditional Lisp systems. Proposals for implementation of the latter form of reflection have been given by Talcott [1983] and by Friedman and Wand [1984]. We plan to use *Rum* as a foundation for studying properties of reflection principles used as programming tools.

Directions for further development of *Rum*. The following questions indicate directions for further work in *Rum*.

- Can the notion of comparison relation be extended to sequential computation? If so, are there analogs to the extensionality and improved recursion theorems?
- What rules should be added to the rules for call-by-value lambda calculus to obtain a *Rum* calculus with reasonable properties? Such a calculus should have the Church-Rosser property and the usual equations for conditional and sequences should be derivable. What happens to such a calculus when extensionality or fixed point induction is added?
- Can the notion of computational progress (Scherlis [1980], see §I.3) be extended to *Rum* to provide a basis for a formal system of program transformations? What are additional interesting examples of transformations involving the use of pfns and continuations? For example, are there systematic methods for deriving co-routines using program transformations in *Rum*?

Appendix A. Rum reference manual

This appendix contains a description of the underlying algebraic structure of *Rum*. Equations are given specifying the domains of t-*Rum* and of s-*Rum*, and the basic operations and computation relations are defined on these domains. Some theorems are stated to indicate key properties of operations not previously defined precisely and to provide precise formulations of some remarks made in the main text. It is assumed that the reader is familiar with the mathematical structures and notation given in Chapter III.

A.1. The domains of *Rum*

The following table lists the domains of *Rum*. The name, domain symbol and a brief description of each domain are given. If there is a set of variables designated to range over a domain this is indicated in the table by < varsym >∈< domsym >. The set of variables consists of subscripted forms of <varsym>. Thus $\mathfrak{f}$, $\mathfrak{f}_0$, $\mathfrak{f}_{\mathrm{arg}}$, ... range over $\mathbb{F}$.

Name	Notation	Description
Data	$\mathbb{D}$	domain of the given data structure
	$d \in \mathbb{D}^*$	sequences of data
Operations	$o \in \mathbb{O}$	operations of the given data structure
Symbols	$s \in \mathbb{S}y$	for naming values
Forms	$\mathfrak{f} \in \mathbb{F}$	symbolic descriptions of computation
Environments	$\xi \in \mathbb{E}$	finite maps binding values to symbols
Dtrees	$\eth \in \mathbb{D}t$	description trees
	$\eth \in \mathbb{D}t^n$	n-ary dtree contexts
Binding Indices	$\chi \in \mathbb{I}x$	finite maps associating numbers to symbols
Pfns	$\varphi \in \mathbb{P}$	descriptions of partial functions
Computation Domain	$a \in \mathbb{V}$	data, data operations, pfns, continuations
Values	$u, v \in \mathbb{V}^*$	sequences from the computation domain
Stages	$\tau \in \mathbb{C}s$	partial computation trees
Nodes	$\nu \in \mathbb{N}^*$	paths or nodes in a tree domain
Continuations	$\gamma \in \mathbb{C}o$	descriptions of computation contexts
	$\mathbb{C}o^s$	continuation segments
States	$\varsigma \in \mathbb{S}t$	stages of sequential computation

The domains of *Rum* are specified by the domain equations given below. The domains are generated freely from $\mathbb{N}$ and $\mathbb{S}y$ by constructions appearing in summands of the corresponding equations. The summands indicate the name and arity of the constructors. Infix notation is used for the summands if infix notation is used for corresponding terms. Objects of s-*Rum* are generated by including summands enclosed in brackets $[\oplus \ldots]$ and objects of t-*Rum* are generated by omitting these summands.

$$\begin{aligned}\mathbb{F} \sim {} & \mathbb{S}y \oplus \lambda(\mathbb{S}y, \mathbb{F}) \oplus \mathsf{app}(\mathbb{F}, \mathbb{F}) \oplus \mathsf{if}(\mathbb{F}, \mathbb{F}, \mathbb{F}) \oplus \mathsf{mt} \oplus \mathsf{cart}(\mathbb{F}, \mathbb{F}) \oplus \mathsf{fst}(\mathbb{F}) \oplus \mathsf{rst}(\mathbb{F}) \\ & [\oplus\ \mathsf{note}(\mathbb{S}y, \mathbb{F})]\end{aligned}$$

$$\begin{aligned}\mathbb{D}t^{n} \sim {} & \triangleleft(\mathbf{V}^{*}) \oplus \#(\{j \in \mathbb{N} \mid j < n\}) \oplus \lambda(\mathbb{D}t^{n+1}) \oplus \mathsf{app}(\mathbb{D}t^{n}, \mathbb{D}t^{n}) \\ & \oplus \mathsf{If}(\mathbb{D}t^{n}, \mathbb{D}t^{n}, \mathbb{D}t^{n}) \oplus \mathsf{mt} \oplus \mathsf{cart}(\mathbb{D}t^{n}, \mathbb{D}t^{n}) \oplus \mathsf{fst}(\mathbb{D}t^{n}) \oplus \mathsf{rst}(\mathbb{D}t^{n}) \\ & [\oplus\ \mathsf{note}(\mathbb{D}t^{n+1})]\end{aligned}$$

$$\begin{aligned}\mathbb{C}o^{s} \sim {} & \mathsf{Ifi}(\mathbb{D}t^{0}, \mathbb{D}t^{0}) \oplus \mathsf{Appi}(\mathbb{D}t^{0}) \oplus \mathsf{Appc}(\mathbf{V}^{*}) \\ & \oplus \mathsf{Carti}(\mathbb{D}t^{0}) \oplus \mathsf{Cartc}(\mathbf{V}^{*}) \oplus \mathsf{Fstc} \oplus \mathsf{Rstc}\end{aligned}$$

$$\mathbb{C}o \sim \mathsf{Id} \oplus (\mathbb{C}o \circ \mathbb{C}o^{s})$$

$$\mathbb{S}t \sim (\mathbb{C}o \triangledown \mathbb{D}t) \oplus (\mathbb{C}o \vartriangle \mathbf{V}^{*})$$

$$\mathbb{D}t \sim \mathbb{D}t^{0} \oplus \mathbb{O}\,'\,\mathbf{D}^{*} \quad [\oplus\ \mathbb{C}o\,'\,\mathbf{V}^{*}]$$

$$\mathbb{P} \sim \lambda(\mathbb{D}t^{1})$$

$$\mathbf{V} \sim \mathfrak{D} \oplus \mathbb{O} \oplus \mathbb{P} \quad [\oplus\ \mathbb{C}o]$$

$$\mathbb{C}s \sim \mathbf{T} : \langle \mathbb{D}t, \mathbf{V}^{*} \rangle$$

$$\mathbf{E} \sim [\mathbb{S}y \ast\!\succ \mathbf{V}^{*}]_{\square}$$

$$\mathbb{L}x \sim [\mathbb{S}y \ast\!\succ \mathbb{N}]$$

$$\mathbb{D}t^{\omega} \sim \cup\{\mathbb{D}t^{n} \mid n \in \mathbb{N}\}$$

Remarks.

- The same names have been used for form and dtree constructors. Context will always make it clear which construction is meant when it matters.

- Dtree contexts $\mathbb{D}t^{\omega}$ are based on the lambda calculus notation of deBruijn [1972].

- The *Rum* domains constitute a many sorted intial algebra (Goguen and Meseguer [1983]). The only objects are those generated by a finite sequence of constructions – no junk, and two objects are equal iff they are in the same domain and have the same construction (modulo rules for equality in the case of finite sequences and finite maps) – no confusion.

- For a complete data structure we would add operations to serve as recognizers (characteristic functions) for each construction and operations for selecting components. These are left implicit in the presentation of *Rum*.

A.2. Summary of operations and relations

The following table lists the operations and relations (other than domain construction operations) defined in this appendix together with their types.

Name	Notation and arity
Index extension	$\wr \in [\mathbb{Ix} \times \mathbb{Sy} \rightarrow \mathbb{Ix}]$
Indexed Closure	$\langle \ldots \mid \ldots ; \ldots \rangle \in [\mathbb{F} \times \mathbb{E} \times \mathbb{Ix} \rightarrow \mathbb{Dt}^{\omega}]$
Alpha Equivalence	$\overset{\alpha}{=} \in [(\mathbb{F}, \mathbb{E}, \mathbb{Ix}) \times (\mathbb{F}, \mathbb{E}, \mathbb{Ix})]$
Substitution	$dsub \in [\mathbb{Dt}^{\omega} \times \mathbb{Dt}^{0} \times \mathbb{N} \rightarrow \mathbb{Dt}^{\omega}]$
Initial Stage	$\bullet \in [\mathbb{Dt} \rightarrow \mathbb{Cs}]$
Begin	$\triangledown \in [\mathbb{Cs} \times \mathbb{N}^{*} \times \mathbb{N} \times \mathbb{Dt} \rightsquigarrow \mathbb{Cs}]$
Return	$\vartriangle \in [\mathbb{Cs} \times \mathbb{N}^{*} \times \mathbf{V}^{*} \rightsquigarrow \mathbb{Cs}]$
Single Step on stages	$\rightarrowtail_{\iota} \subset [\mathbb{Cs} \times \mathbb{Cs}]$
Steps on stages	$\rightarrowtail, \overset{+}{\rightarrowtail} \subset [\mathbb{Cs} \times \mathbb{Cs}]$
Evaluation	$\hookrightarrow \in [\mathbb{Dt} \rightsquigarrow \mathbf{V}^{*}]$
Immediate Subcomputation	$\prec_{\iota} \subset [\mathbb{Dt} \times \mathbb{Dt}]$
Immediately Reduces-to	$\succ\!\!\succ_{\iota} \in [\mathbb{Dt} \rightsquigarrow \mathbb{Dt}]$
Subcomputation	$\prec \subset [\mathbb{Dt} \times \mathbb{Dt}]$
Reduces-to	$\succ\!\!\succ \subset [\mathbb{Dt} \times \mathbb{Dt}]$
Definedness	$\Downarrow \subset \mathbb{Dt}$
Ctree equivalence	$\rightleftharpoons \subset [\mathbb{Dt} \times \mathbb{Dt}]$
Single Step on states	$\rightarrowtail_{\iota} \in [\mathbb{St} \rightsquigarrow \mathbb{St}]$
Steps on states	$\rightarrowtail, \overset{+}{\rightarrowtail} \subset [\mathbb{St} \times \mathbb{St}]$
Cseq equivalence	$\approx \subset [\mathbb{St} \times \mathbb{St}]$
Returns to caller	$\downarrow \subset \mathbb{Dt}$
Escapes to top	${}^{\mathsf{Id}}\!\downarrow \subset \mathbb{Dt}$
Diverges	$\uparrow \subset \mathbb{Dt}$
Locality	$\uparrow\!\downarrow \subset \mathbb{Dt}$

A.3. Closure and substitution operations

▷ **Extending index maps.** $\chi \wr s \underset{\mathrm{df}}{=} \{s \leftarrow 0\} \cup \{s_0 \leftarrow \chi(s_0) + 1 \mid s_0 \in dom(\chi)\}$

▷ **Indexed closure operation.** $\langle \mathfrak{f} \mid \xi; \chi \rangle$ is defined by induction on forms.

$$\langle s \mid \xi; \chi \rangle = \begin{cases} \#(\chi(s)) & \text{if } s \in dom(\chi) \\ \triangleleft(\xi(s)) & \text{otherwise} \end{cases}$$

$$\langle \lambda(s)\mathfrak{f} \mid \xi; \chi \rangle = \lambda(\langle \mathfrak{f} \mid \xi; \chi \wr s \rangle)$$

$$\langle \mathsf{app}(\mathfrak{f}_0, \mathfrak{f}_1) \mid \xi; \chi \rangle = \mathsf{app}(\langle \mathfrak{f}_0 \mid \xi; \chi \rangle, \langle \mathfrak{f}_1 \mid \xi; \chi \rangle)$$

$$\langle \mathsf{if}(\mathfrak{f}_0, \mathfrak{f}_1, \mathfrak{f}_2) \mid \xi; \chi \rangle = \mathsf{if}(\langle \mathfrak{f}_0 \mid \xi; \chi \rangle, \langle \mathfrak{f}_1 \mid \xi; \chi \rangle, \langle \mathfrak{f}_2 \mid \xi; \chi \rangle)$$

$$\langle \mathsf{mt} \mid \xi; \chi \rangle = \mathsf{mt}$$

$$\langle \mathsf{cart}(\mathfrak{f}_0, \mathfrak{f}_1) \mid \xi; \chi \rangle = \mathsf{cart}(\langle \mathfrak{f}_0 \mid \xi; \chi \rangle, \langle \mathfrak{f}_1 \mid \xi; \chi \rangle)$$

$$\langle \mathsf{fst}(\mathfrak{f}) \mid \xi; \chi \rangle = \mathsf{fst}(\langle \mathfrak{f} \mid \xi; \chi \rangle)$$

$$\langle \mathsf{rst}(\mathfrak{f}) \mid \xi; \chi \rangle = \mathsf{rst}(\langle \mathfrak{f} \mid \xi; \chi \rangle)$$

$$\langle \mathsf{note}(s)\mathfrak{f} \mid \xi; \chi \rangle = \mathsf{note}(\langle \mathfrak{f} \mid \xi; \chi \wr s \rangle)$$

▷ **Closure.** $\langle \mathfrak{f} \mid \xi \rangle \underset{\mathsf{df}}{=} \langle \mathfrak{f} \mid \xi; \{\} \rangle$

- **Example.**

$$\begin{aligned} &\langle \lambda(\mathsf{y})\mathsf{if}(\mathsf{y}, \lambda(\mathsf{x})\mathsf{x}(\mathsf{y}), \mathsf{z}) \mid \mathsf{z} \leftarrow v \rangle = \lambda(\langle \mathsf{if}(\mathsf{y}, \lambda(\mathsf{x})\mathsf{x}(\mathsf{y}), \mathsf{z}) \mid \mathsf{z} \leftarrow v; \mathsf{y} \leftarrow 0 \rangle) \\ &\quad = \lambda(\mathsf{if}(\langle \mathsf{y} \mid \mathsf{z} \leftarrow v; \mathsf{y} \leftarrow 0 \rangle, \langle \lambda(\mathsf{x})\mathsf{x}(\mathsf{y}) \mid \mathsf{z} \leftarrow v; \mathsf{y} \leftarrow 0 \rangle, \langle \mathsf{z} \mid \mathsf{z} \leftarrow v; \mathsf{y} \leftarrow 0 \rangle)) \\ &\quad = \lambda(\mathsf{if}(\#0, \lambda(\#0(\#1)), \triangleleft v)) \end{aligned}$$

▷ **Alpha equivalence.** Alpha equivalence is the *Rum* analog to the equivalence relation generated by α-conversion in the lambda calculus. Alpha equivalence on form - environment - index triples is the least relation $\overset{\alpha}{=}$ such that

$$s_0 \in dom(\chi_0) \land s_1 \in dom(\chi_1) \land \chi_0(s_0) = \chi_1(s_1) \rightarrow (s_0, \xi_0, \chi_0) \overset{\alpha}{=} (s_1, \xi_1, \chi_1)$$

$$s_0 \notin dom(\chi_0) \land s_1 \notin dom(\chi_1) \land \xi_0(s_0) = \xi_1(s_1) \rightarrow (s_0, \xi_0, \chi_0) \overset{\alpha}{=} (s_1, \xi_1, \chi_1)$$

$$(\mathsf{mt}, \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{mt}, \xi_1, \chi_1)$$

$$(\mathfrak{f}_{0,\mathrm{i}}, \xi_0, \chi_0) \overset{\alpha}{=} (\mathfrak{f}_{1,\mathrm{i}}, \xi_1, \chi_1), 0 \leq i \leq 2 \rightarrow$$

$$\begin{aligned} &(\mathsf{if}(\mathfrak{f}_{0,0}, \mathfrak{f}_{0,1}, \mathfrak{f}_{0,2}), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{if}(\mathfrak{f}_{1,0}, \mathfrak{f}_{0,1}, \mathfrak{f}_{1,2}), \xi_1, \chi_1) \land \\ &(\mathsf{app}(\mathfrak{f}_{0,0}, \mathfrak{f}_{0,1}), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{app}(\mathfrak{f}_{1,0}, \mathfrak{f}_{0,1}), \xi_1, \chi_1) \land \\ &(\mathsf{cart}(\mathfrak{f}_{0,0}, \mathfrak{f}_{0,1}), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{cart}(\mathfrak{f}_{1,0}, \mathfrak{f}_{0,1}), \xi_1, \chi_1) \land \\ &(\mathsf{fst}(\mathfrak{f}_{0,0}), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{fst}(\mathfrak{f}_{1,0}), \xi_1, \chi_1) \land \\ &(\mathsf{rst}(\mathfrak{f}_{0,0}), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{rst}(\mathfrak{f}_{1,0}), \xi_1, \chi_1) \end{aligned}$$

$$(\mathfrak{f}_0, \xi_0, \chi_0 \wr s_0) \overset{\alpha}{=} (\mathfrak{f}_1, \xi_1, \chi_1 \wr s_1) \rightarrow$$

$$\begin{aligned} &(\lambda(s_0)\mathfrak{f}_0), \xi_0, \chi_0) \overset{\alpha}{=} (\lambda(s_1)\mathfrak{f}_1), \xi_1, \chi_1) \land \\ &(\mathsf{note}(s_0)\mathfrak{f}_0), \xi_0, \chi_0) \overset{\alpha}{=} (\mathsf{note}(s_1)\mathfrak{f}_1), \xi_1, \chi_1) \end{aligned}$$

▪ **Alpha equivalence is dtree equality.**

$$(\mathfrak{f}_0,\xi_0,\chi_0) \overset{\alpha}{=} (\mathfrak{f}_1,\xi_1,\chi_1) \leftrightarrow \langle \mathfrak{f}_0 \mid \xi_0;\chi_0\rangle = \langle \mathfrak{f}_1 \mid \xi_1;\chi_1\rangle$$

▹ **Substitution of dtrees into dtree contexts.** *dsub* is defined by induction on dtree contexts.

$$dsub(\#j,\mathfrak{d},n) = \begin{cases} \mathfrak{d} & \text{if } j = n \\ \#j & \text{otherwise} \end{cases}$$

$$dsub(\lhd u,\mathfrak{d},n) = \lhd u$$

$$dsub(\lambda(\mathfrak{d}_0),\mathfrak{d},n) = \lambda(dsub(\mathfrak{d}_0,\mathfrak{d},n+1))$$

$$dsub(\mathsf{app}(\mathfrak{d}_0,\mathfrak{d}_1),\mathfrak{d},n) = \mathsf{app}(dsub(\mathfrak{d}_0,\mathfrak{d},n),dsub(\mathfrak{d}_1,\mathfrak{d},n))$$

$$dsub(\mathsf{if}(\mathfrak{d}_0,\mathfrak{d}_1,d_2),\mathfrak{d},n) = \mathsf{if}(dsub(\mathfrak{d}_0,\mathfrak{d},n),dsub(\mathfrak{d}_1,\mathfrak{d},n),dsub(\mathfrak{d}_2,\mathfrak{d},n))$$

$$dsub(\mathsf{mt},\mathfrak{d},n) = \mathsf{mt}$$

$$dsub(\mathsf{cart}(\mathfrak{d}_0,\mathfrak{d}_1),\mathfrak{d},n) = \mathsf{cart}(dsub(\mathfrak{d}_0,\mathfrak{d},n),dsub(\mathfrak{d}_1,\mathfrak{d},n))$$

$$dsub(\mathsf{fst}(\mathfrak{d}_0),\mathfrak{d},n) = \mathsf{fst}(dsub(\mathfrak{d}_0,\mathfrak{d},n))$$

$$dsub(\mathsf{rst}(\mathfrak{d}_0),\mathfrak{d},n) = \mathsf{rst}(dsub(\mathfrak{d}_0,\mathfrak{d},n))$$

$$dsub(\mathsf{note}(\mathfrak{d}_0),\mathfrak{d},n) = \mathsf{note}(dsub(\mathfrak{d}_0,\mathfrak{d},n+1))$$

Remark. A more complicated definition is required for substitution of dtree contexts into dtree contexts.

▹ **Extending application and substitution notation.**

$$\lambda(\mathfrak{d}_0)'v \underset{\mathrm{df}}{=} dsub(\mathfrak{d}_0,\lhd v,0)$$

$$\lambda(\mathfrak{d}_0)'\mathfrak{d} \underset{\mathrm{df}}{=} dsub(\mathfrak{d}_0,\mathfrak{d},0)$$

$$\mathfrak{d}\{\mathfrak{d}_0,\ldots,\mathfrak{d}_{\mathrm{m}}\} \underset{\mathrm{df}}{=} dsub(\ldots..dsub(\mathfrak{d},\mathfrak{d}_{\mathrm{m}},m)\ldots,\mathfrak{d}_0,0)$$

▹ **Well-formed application expression.** It is now convenient to think of application as a binary operation on values. We say that $v_0{}'v_1$ is *well-formed* iff $v_0 \in \mathbb{O} \wedge v_1 \in \mathfrak{D}^*$ or $v_0 \in \mathbb{P}$ or $v_0 \in \mathbb{C}_o$.

▪ **About substitution and application.**

$$(\lambda(\mathfrak{d})\{\mathfrak{d}_0\ldots\mathfrak{d}_{\mathrm{m}}\})'v = \mathfrak{d}\{\lhd v,\mathfrak{d}_0\ldots\mathfrak{d}_{\mathrm{m}}\}$$

$$\langle \mathfrak{f}|^{s}_{\mathfrak{f}_0} \mid \xi\rangle = dsub(\langle \mathfrak{f} \mid \xi;\{s \leftarrow 0\}\rangle,\langle \mathfrak{f}_0 \mid \xi\rangle,0)$$

$$dsub(\langle \mathfrak{f} \mid \xi;\chi\rangle,\lhd v,n) = \langle \mathfrak{f} \mid \xi_0;\chi_0\rangle$$

$$\text{where} \quad \xi_0 = \xi\{s \leftarrow v \mid \chi(s) = n\} \quad \text{and} \quad \chi_0 = \{s \leftarrow \chi(s) \mid s \in dom(\chi) \wedge \chi(s) \neq n\}$$

$$(\forall s \in dom(\chi))\chi(s) \neq n \rightarrow dsub(\langle \mathfrak{f} \mid \xi;\chi\rangle,\mathfrak{d},n) = \langle \mathfrak{f} \mid \xi;\chi\rangle$$

$$\langle \lambda(s)\mathfrak{f}) \mid \xi\rangle'v = \langle \mathfrak{f} \mid \xi\{s \leftarrow v\}\rangle$$

A.4. Generating computation trees

▷ **Initial stage.** $\bullet\partial \underset{\text{df}}{=} <\{\square\},\{\square \leftarrow \partial\},\{\}>$

▷ **Extending stages.** For a stage $\tau = <\Delta, \Lambda^d, \Lambda^v>$ and $\nu \in \Delta$ define

(begin) $\tau \triangledown (\nu, j, \partial) \underset{\text{df}}{=} <\Delta \cup \{[\nu, j]\}, \Lambda^d\{[\nu, j] \leftarrow \partial\}, \Lambda^v>$

(return) $\tau \vartriangle (\nu, v) \underset{\text{df}}{=} <\Delta, \Lambda^d, \Lambda^v\{\nu \leftarrow v\}>$

▷ **Step rules.** $\mathit{Step.rules} \underset{\text{df}}{=} \{(\nu, j, \partial) \in [\mathbb{N}^* \times \mathbb{N} \times \mathbb{D}t]\} \cup \{(\nu, v) \in [\mathbb{N}^* \times \mathbf{V}^*]\}$

▷ **When a step rule applies.**[†] The conditions under which a step rule applies to a stage $<\Delta, \Lambda^d, \Lambda^v>$ are defined by

$$
\begin{aligned}
&\mathit{applies}((\nu, j, \partial), <\Delta, \Lambda^d, \Lambda^v>) \leftrightarrow \nu \in \Delta \wedge [\nu, j] \notin \Delta \wedge \\
&\quad (\exists \partial_0, \partial_1)(\Lambda^d(\nu) = \mathsf{app}(\partial_0, \partial_1) \wedge \\
&\qquad ((j = 0 \wedge \partial = \partial_0) \vee (j = 1 \wedge \partial = \partial_1) \vee (j = 2 \wedge \partial = \Lambda^v([\nu, 0])\,'\,\Lambda^v([\nu, 1])))) \vee \\
&\quad (\exists \partial_0, \partial_1)(\Lambda^d(\nu) = \mathsf{if}(\partial_0, \partial_1, \partial_2) \wedge \\
&\qquad \left((j = 0 \wedge \partial = \partial_0) \vee (j = 1 \wedge \partial = \begin{cases} \partial_1 & \text{if } \Lambda^v([\nu, 0]) \neq \square \\ \partial_2 & \text{if } \Lambda^v([\nu, 0]) = \square \end{cases}))\right) \vee \\
&\quad (\exists \partial_0, \partial_1)(\Lambda^d(\nu) = \mathsf{cart}(\partial_0, \partial_1) \wedge ((j = 0 \wedge \partial = \partial_0) \vee (j = 1 \wedge \partial = \partial_1))) \vee \\
&\quad (\exists \partial_0)(\Lambda^d(\nu) = \mathsf{fst}(\partial_0) \wedge j = 0 \wedge \partial = \partial_0) \vee \\
&\quad (\exists \partial_0)(\Lambda^d(\nu) = \mathsf{rst}(\partial_0) \wedge j = 0 \wedge \partial = \partial_0)
\end{aligned}
$$

and

$$
\begin{aligned}
&\mathit{applies}((\nu, v), <\Delta, \Lambda^d, \Lambda^v>) \leftrightarrow \nu \in \Delta_0 \wedge \nu \notin \mathit{dom}(\Lambda^v_0) \wedge \\
&\quad (\exists o, d)(\Lambda^d(\nu) = o\,'\,d \wedge v = o(d)) \vee \Lambda^d(\nu) = \lhd v \vee \Lambda^d(\nu) = \mathsf{mt} \wedge v = \square \vee \\
&\quad (\exists \partial_0)(\Lambda^d(\nu) = \lambda(\partial_0) \wedge v = \lambda(\partial_0)) \vee \\
&\quad (\exists \partial_0, \partial_1)(\Lambda^d(\nu) = \mathsf{app}(\partial_0, \partial_1) \wedge v = \Lambda^v([\nu, 2])) \vee \\
&\quad (\exists \partial_0, \partial_1, \partial_2)(\Lambda^d(\nu) = \mathsf{if}(\partial_0, \partial_1, \partial_2) \wedge v = \Lambda^v([\nu, 1])) \vee \\
&\quad (\exists \partial_0, \partial_1)(\Lambda^d(\nu) = \mathsf{cart}(\partial_0, \partial_1) \wedge v = [\Lambda^v([\nu, 0]), \Lambda^v([\nu, 1])]) \vee \\
&\quad (\exists \partial_0)(\Lambda^d(\nu) = \mathsf{fst}(\partial_0) \wedge v = 1^{st}\,(\Lambda^v([\nu, 0]))) \vee \\
&\quad (\exists \partial_0)(\Lambda^d(\nu) = \mathsf{rst}(\partial_0) \wedge v = r^{st}\,(\Lambda^v([\nu, 0])))
\end{aligned}
$$

[†] Implicit in the definitions are hypotheses that arguments are in the domain of partial operations. For example, implicit in $v = \Lambda^v([\nu, 1]))$ is the assertion that $[\nu, 1] \in \mathit{dom}(\Lambda^v)$.

▷ **Computation steps.** For stages τ_i

$$\tau_0 \rightarrowtail_\iota \tau_1 \underset{\mathrm{df}}{\equiv} (\exists \nu, j, \mathfrak{d})(\tau_0 \overset{(\nu,j,\mathfrak{d})}{\rightarrowtail} \tau_1) \vee (\exists \nu, v)(\tau_0 \overset{(\nu,v)}{\rightarrowtail} \tau_1)$$

$$\tau_0 \overset{(\nu,j,\mathfrak{d})}{\rightarrowtail} \tau_1 \underset{\mathrm{df}}{\equiv} \tau_1 = \tau_0 \triangledown (\nu, j, \mathfrak{d}) \wedge \mathit{applies}((\nu, j, \mathfrak{d}), \tau_0)$$

$$\tau_0 \overset{(\nu,v)}{\rightarrowtail} \tau_1 \underset{\mathrm{df}}{\equiv} \tau_1 = \tau_0 \vartriangle (\nu, v) \wedge \mathit{applies}((\nu, v), \tau_0)$$

$$\rightarrowtail \underset{\mathrm{df}}{=} (\rightarrowtail_\iota)^*$$

▷ **The computation tree for $\mathfrak{d}$.** $Ct(\mathfrak{d}) = <\Delta_\mathfrak{d}, \Lambda_\mathfrak{d}^d, \Lambda_\mathfrak{d}^v> \underset{\mathrm{df}}{=} \bigcup\{\tau \mid \bullet\mathfrak{d} \rightarrowtail \tau\}$

■ **Small diamond property.**

$$\{\mathrm{r1},\ \mathrm{r2}\} \subset \mathit{Step.rule} \wedge \tau_0 \overset{r1}{\rightarrowtail} \tau_1 \wedge \tau_0 \overset{r2}{\rightarrowtail} \tau_2 \rightarrow (\exists \tau_3)(\tau_1 \overset{r2}{\rightarrowtail} \tau_3 \wedge \tau_2 \overset{r1}{\rightarrowtail} \tau_3)$$

■ **Steps increase stages.** $\tau_0 \rightarrowtail \tau_1 \rightarrow \tau_0 \subset \tau_1$

■ **Steps in substages lift.** $\tau{\downarrow_\nu} = \tau_0 \wedge \tau_0 \rightarrowtail \tau_1 \rightarrow \tau \rightarrowtail \tau\{\nu \leftarrow \tau_1\}$

■ **Subtrees.** $\nu \in \Delta_\mathfrak{d} \wedge \Lambda_\mathfrak{d}^d(\nu) = \mathfrak{d}_0 \rightarrow Ct(\mathfrak{d}){\downarrow_\nu} = Ct(\mathfrak{d}_0)$

■ **A computation that returns a value is finite.** $\square \in \mathit{dom}(\Lambda_\mathfrak{d}^v) \rightarrow \bullet\mathfrak{d} \rightarrowtail Ct(\mathfrak{d})$

A.5. Computation relations for tree-structured computation

▷ **Evaluation, reduces-to, subcomputation.** Evaluation ($\hookrightarrow$), immediately reduces-to ($\succ\!\!\succ_\iota$), and immediate subcomputation ($\prec_\iota$) are the least relations such that

(val) $\triangleleft v \hookrightarrow v$

(lam) $\lambda(\mathfrak{d}) \hookrightarrow \lambda(\mathfrak{d})$

(dapp) $o\,'\,d \hookrightarrow o(d)$

(mt) $\mathsf{mt} \hookrightarrow \square$

(cart) $\mathfrak{d}_0 \hookrightarrow v_0 \wedge \mathfrak{d}_1 \hookrightarrow v_1 \rightarrow \mathsf{cart}(\mathfrak{d}_0, \mathfrak{d}_) \hookrightarrow [v_0, v_1]$

(fst) $\mathfrak{d}_0 \hookrightarrow v_0 \rightarrow \mathsf{fst}(\mathfrak{d}_0) \hookrightarrow 1^{st}(v_0)$

(rst) $\mathfrak{d}_0 \hookrightarrow v_0 \rightarrow \mathsf{rst}(\mathfrak{d}_0) \hookrightarrow r^{st}(v_0)$

(redt.ret) $\mathfrak{d} \succ\!\!\succ_\iota \mathfrak{d}_0 \wedge \mathfrak{d}_0 \hookrightarrow v \rightarrow \mathfrak{d} \hookrightarrow v$

(app) $\mathfrak{d}_0 \hookrightarrow v_0 \wedge \mathfrak{d}_1 \hookrightarrow v_1 \rightarrow \mathsf{app}(\mathfrak{d}_0, \mathfrak{d}_1) \succ\!\!\succ_\iota v_0\,'\,v_1$ % $v_0\,'\,v_1$ well-formed

(if) $\mathfrak{d}_0 \hookrightarrow v_0 \rightarrow \mathsf{if}(\mathfrak{d}_0, \mathfrak{d}_1, \mathfrak{d}_2) \succ\!\!\succ_\iota \begin{cases} \mathfrak{d}_1 & \text{if } v_0 \neq \square \\ \mathfrak{d}_2 & \text{if } v_0 = \square \end{cases}$

(if.test) $\mathfrak{d}_0 \prec_\iota \mathsf{if}(\mathfrak{d}_0, \mathfrak{d}_1, \mathfrak{d}_2)$

(app.fun) $\mathfrak{d}_0 \prec_\iota \mathsf{app}(\mathfrak{d}_0, \mathfrak{d}_1)$

(app.arg) $\mathfrak{d}_1 \prec_\iota \mathsf{app}(\mathfrak{d}_0, \mathfrak{d}_1)$

(cart.lhs) $\mathfrak{d}_0 \prec_\iota \mathsf{cart}(\mathfrak{d}_0, \mathfrak{d}_1)$

(cart.rhs) $\mathfrak{d}_1 \prec_\iota \mathsf{cart}(\mathfrak{d}_0, \mathfrak{d}_1)$

(fst.seq) $\mathfrak{d}_0 \prec_\iota \mathsf{fst}(\mathfrak{d}_0)$

(rst.seq) $\mathfrak{d}_0 \prec_\iota \mathsf{rst}(\mathfrak{d}_0)$

(redt.beg) $\mathfrak{d} \succ\!\!\succ_\iota \mathfrak{d}_0 \rightarrow \mathfrak{d}_0 \prec_\iota \mathfrak{d}$

□end definition of $\hookrightarrow$, $\succ\!\!\succ_\iota$, $\prec_\iota$

▷ **Additional t-*Rum* relations**

$$\succ\!\!\succ \underset{\mathrm{df}}{\equiv} {\succ\!\!\succ_\iota}^{+} \qquad \prec \underset{\mathrm{df}}{\equiv} {\prec_\iota}^{+}$$

$$\downdownarrows\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\exists v)\mathfrak{d} \hookrightarrow v \qquad \mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} (\forall v)(\mathfrak{d}_0 \hookrightarrow v \leftrightarrow \mathfrak{d}_1 \hookrightarrow v)$$

■ **Equivalence of computation trees and computation relations.**

(eval) $\mathfrak{d} \hookrightarrow v \leftrightarrow \Lambda^{v}_{\mathfrak{d}}(\square) = v$

(subc) $\mathfrak{d}_0 \prec \mathfrak{d} \leftrightarrow (\exists\langle\Delta, \Lambda^{d}, \Lambda^{v}\rangle, \nu, j)(\bullet\mathfrak{d} \rightarrowtail \langle\Delta, \Lambda^{d}, \Lambda^{v}\rangle \wedge \Lambda^{d}([\nu, j]) = \mathfrak{d}_0)$

A.6. Sequential computation

▷ **Single step for computation states.** The single step relation ($\rightarrowtail_\iota$) is the least relation on $[\mathbb{S}\mathrm{t} \times \mathbb{S}\mathrm{t}]$ satisfying

(val) $\gamma \triangledown \triangleleft v \rightarrowtail_\iota \gamma \vartriangle v$

(lam) $\gamma \triangledown \lambda(\mathfrak{e}) \rightarrowtail_\iota \gamma \vartriangle \lambda(\mathfrak{e})$

(dapp) $\gamma \triangledown o' d \rightarrowtail_\iota \gamma \vartriangle o(d)$

(capp) $\gamma \triangledown \gamma_0{}' v \rightarrowtail_\iota \gamma_0 \vartriangle v$

(mt) $\gamma \triangledown \mathsf{mt} \rightarrowtail_\iota \gamma \vartriangle \square$

(cart.ret) $\gamma \circ \mathsf{Cartc}(v_0) \vartriangle v_1 \rightarrowtail_\iota \gamma \vartriangle [v_0, v_0]$

(fst.ret) $\gamma \circ \mathsf{Fstc} \vartriangle v \rightarrowtail_\iota \gamma \vartriangle 1^{\mathrm{st}}(v)$

(rst.ret) $\gamma \circ \mathsf{Rstc} \vartriangle v \rightarrowtail_\iota \gamma \vartriangle \mathrm{r}^{\mathrm{st}}(v)$

(app.fun) $\gamma \triangledown \mathsf{app}(\mathfrak{e}_0, \mathfrak{e}_1) \rightarrowtail_\iota \gamma \circ \mathsf{Appl}(\mathfrak{e}_1) \triangledown \mathfrak{e}_0$

(app.arg) $\gamma \circ \mathsf{Appl}(\mathfrak{e}_1) \vartriangle v_0 \rightarrowtail_\iota \gamma \circ \mathsf{Appc}(v_0) \triangledown \mathfrak{e}_1$

(app.app) $\gamma \circ \mathsf{Appc}(v_0) \vartriangle v_1 \rightarrowtail_\iota \gamma \triangledown v_0{}' v_1$ % if $v_0{}' v_1$ is well-formed

(if.test) $\gamma \triangledown \mathsf{if}(\mathfrak{e}_0, \mathfrak{e}_1, \mathfrak{e}_2) \rightarrowtail_\iota \gamma \circ \mathsf{Ifi}(\mathfrak{e}_1, \mathfrak{e}_2) \triangledown \mathfrak{e}_0$

(if.br) $\gamma \circ \mathsf{Ifi}(\mathfrak{e}_1, \mathfrak{e}_2) \vartriangle v_0 \rightarrowtail_\iota \gamma \triangledown \begin{cases} \mathfrak{e}_1 & \text{if } v_0 \neq \square \\ \mathfrak{e}_2 & \text{if } v_0 = \square \end{cases}$

(cart.lhs) $\gamma \triangledown \mathsf{cart}(\mathfrak{e}_0, \mathfrak{e}_1) \rightarrowtail_\iota \gamma \circ \mathsf{Cartl}(\mathfrak{e}_1) \triangledown \mathfrak{e}_0$

(cart.rhs) $\gamma \circ \mathsf{Cartl}(\mathfrak{e}_1) \vartriangle v_0 \rightarrowtail_\iota \gamma \circ \mathsf{Cartc}(v_0) \triangledown \mathfrak{e}_1$

(fst.beg) $\gamma \triangledown \mathsf{fst}(\mathfrak{e}) \rightarrowtail_\iota \gamma \circ \mathsf{Fstc} \triangledown \mathfrak{e}$

(rst.beg) $\gamma \triangledown \mathsf{rst}(\mathfrak{e}) \rightarrowtail_\iota \gamma \circ \mathsf{Rstc} \triangledown \mathfrak{e}$

(note) $\gamma \triangledown \mathsf{note}(\mathfrak{e}) \rightarrowtail_\iota \gamma \triangledown \mathfrak{e}\{\triangleleft\gamma\}$

□end definition of $\rightarrowtail_\iota$

▷ **Step relation for states.** $\rightarrowtail \; = (\rightarrowtail_\iota)^*$ and $\overset{+}{\rightarrowtail} = (\rightarrowtail_\iota)^+$

▷ **Sequential equivalence.** $\mathfrak{e}_0 \approx \mathfrak{e}_1 \underset{\mathrm{df}}{\equiv} (\forall \gamma, v)(\gamma \triangledown \mathfrak{e}_0 \rightarrowtail \mathsf{Id} \vartriangle v \leftrightarrow \gamma \triangledown \mathfrak{e}_1 \rightarrowtail \mathsf{Id} \vartriangle v)$

▷ **Returns to caller.** $\downarrow\mathfrak{e} \underset{\mathrm{df}}{\equiv} (\forall \gamma)(\exists v)\gamma \triangledown \mathfrak{e} \rightarrowtail \gamma \vartriangle v$

▷ **Escapes to top.** ${}^{\mathsf{Id}}{\downarrow}\mathfrak{e} \underset{\mathrm{df}}{\equiv} (\forall \gamma)(\exists v)\gamma \triangledown \mathfrak{e} \rightarrowtail \mathsf{Id} \vartriangle v$

▷ **Diverges.** $\uparrow\mathfrak{e} \underset{\mathrm{df}}{\equiv} \neg(\exists v)(\mathsf{Id} \triangledown \mathfrak{e} \rightarrowtail \mathsf{Id} \vartriangle v)$

▷ **Evaluation in $\mathfrak{s}$-$\mathcal{R}um$.** $\mathfrak{d} \hookrightarrow v \underset{\mathrm{df}}{\equiv} (\forall\gamma)(\gamma \triangledown \mathfrak{d} \rightarrowtail \gamma \vartriangle v)$

▷ **Reduces-to in $\mathfrak{s}$-$\mathcal{R}um$.** $\mathfrak{d}_0 \twoheadrightarrow \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} (\forall\gamma)(\gamma \triangledown \mathfrak{d}_0 \rightarrowtail \gamma \triangledown \mathfrak{d}_1)$

▷ **Definedness in $\mathfrak{s}$-$\mathcal{R}um$.** $\Downarrow\mathfrak{d} \underset{\mathrm{df}}{\equiv} (\exists v)\mathfrak{d} \hookrightarrow v$

▷ **Locality.** $\uparrow\downarrow\mathfrak{d} \underset{\mathrm{df}}{\equiv} \Downarrow\mathfrak{d} \vee \uparrow\mathfrak{d}$

▷ **Ctree equivalence in $\mathfrak{s}$-$\mathcal{R}um$.** $\mathfrak{d}_0 \rightleftharpoons \mathfrak{d}_1 \underset{\mathrm{df}}{\equiv} \uparrow\downarrow\mathfrak{d}_0 \wedge \uparrow\downarrow\mathfrak{d}_1 \wedge (\forall v)(\mathfrak{d}_0 \hookrightarrow v \leftrightarrow \mathfrak{d}_1 \hookrightarrow v)$

Appendix B. More examples

In this appendix we present two examples to illustrate further what can be expressed in *Rum* and to provide applications of the tools developed so far. The first example gives a more precise formulation of intensional properties of programs than we have previously given. The second example illustrates a new use of pfnls and their properties as schemes for defining and proving properties of search pfns.

B.1. Derived computations

In this example a class of properties of computation trees called derived properties and a derivation map on descriptions are defined. Derived properties are given by a scheme for recursion on computation trees parameterized by a sequence of primitive derivations – one for each computation primitive. The derivation map is parameterized by a sequence of derivation symbols naming the primitive derivations. Derived descriptions compute the derived properties of the original computation together with (a derivation of) the original value. Examples of specific derived properties are given, including many of the properties previously discussed informally. As an application, we show that the derived pfn Mtprod of §II.1 can be obtained by specializing the derivation of Tprod to the primitive derivations for counting multiplications.

B.1.1. Derived properties of computation trees

Let $\mathbb{D}_0 \subset \mathbb{D}$ and let ∇ be a sequence from $\mathbf{V}$

$$\nabla = [\nabla_{sym}, \nabla_{mt}, \nabla_{lam}, \nabla_{if}, \nabla_{app}, \nabla_{cart}, \nabla_{fst}, \nabla_{rst}, \nabla_{o} \mid o \in \mathbb{O}]$$

We say that ∇ is a sequence of primitive derivations if

$$\nabla_o \in \mathbb{D}_0 \quad \text{for} \quad o \in \mathbb{O}, \qquad \nabla_{sym} \in \mathbb{D}_0, \qquad \nabla_{lam} \in \mathbb{D}_0, \qquad \nabla_{mt} \in \mathbb{D}_0$$
$$\nabla_{if} \in [\mathbb{D}_0 \to \mathbb{D}_0 \to \mathbb{D}_0], \qquad \nabla_{app} \in [\mathbb{D}_0 \to \mathbb{D}_0 \to \mathbb{D}_0 \to \mathbb{D}_0]$$
$$\nabla_{cart} \in [\mathbb{D}_0 \to \mathbb{D}_0 \to \mathbb{D}_0], \qquad \nabla_{fst} \in [\mathbb{D}_0 \to \mathbb{D}_0], \qquad \nabla_{rst} \in [\mathbb{D}_0 \to \mathbb{D}_0]$$

where for pfns and data operations, $\vartheta \in [\mathbb{D}_0 \to \mathbb{D}_0 \to \mathbb{D}_0]$ means $(\forall d_0, d_1 \in \mathbb{D}_0)(\exists d \in \mathbb{D}_0)\varphi'(\partial_0, d_1) \hookrightarrow d$, etc.

▷ **The computation tree derivation.** For computation trees $Ct(\mathfrak{d})$ such that $\Downarrow\mathfrak{d}$, the derivation $Ct(\mathfrak{d})^{\nabla}$ is defined by induction on the tree structure

$$(Ct(\triangleleft v))^{\nabla} = \nabla_{sym}$$

$$(Ct(\mathsf{mt}))^{\nabla} = \nabla_{mt}$$

$$(Ct(\lambda(\mathfrak{d}_0)))^{\nabla} = \nabla_{lam}$$

$$(Ct(o\,'\,d))^{\nabla} = \nabla_{o}$$

$$(Ct(\mathsf{if}(\mathfrak{d}_0,\mathfrak{d}_1,\mathfrak{d}_2)))^{\nabla} = \nabla_{if}((Ct(\mathfrak{d}_0))^{\nabla}, d_1)$$

$$\text{where} \quad d_1 = \begin{cases} (Ct(\mathfrak{d}_1))^{\nabla} & \text{if } \neg(\mathfrak{d}_0 \hookrightarrow \square) \\ (Ct(\mathfrak{d}_2))^{\nabla} & \text{if } \mathfrak{d}_0 \hookrightarrow \square \end{cases}$$

$$(Ct(\mathsf{app}(\mathfrak{d}_0,\mathfrak{d}_1)))^{\nabla} = \nabla_{app}((Ct(\mathfrak{d}_0))^{\nabla},(Ct(\mathfrak{d}_1))^{\nabla},(Ct(\mathfrak{d}_2))^{\nabla})$$

$$\text{where} \quad \mathfrak{d}_0 \hookrightarrow v_0 \wedge \mathfrak{d}_1 \hookrightarrow v_1 \wedge \mathfrak{d}_2 = v_0\,'\,v_1$$

$$(Ct(\mathsf{cart}(\mathfrak{d}_0,\mathfrak{d}_1)))^{\nabla} = \nabla_{cart}((Ct(\mathfrak{d}_0))^{\nabla},(Ct(\mathfrak{d}_1))^{\nabla})$$

$$(Ct(\mathsf{fst}(\mathfrak{d}_0)))^{\nabla} = \nabla_{fst}((Ct(\mathfrak{d}_0))^{\nabla})$$

$$(Ct(\mathsf{rst}(\mathfrak{d}_0)))^{\nabla} = \nabla_{rst}((Ct(\mathfrak{d}_0))^{\nabla})$$

B.1.2. Example derivations

Working over $\mathfrak{D}_{\mathrm{sexp}}$ we give four examples to indicate a few of the properties of computation trees that can be expressed by the derivation just defined.

Example 1: Counting data operations. Let O be a set of data operations. In §II.1 we introduced $count(\mathfrak{d}, O)$, the number of data operation application nodes in $Ct(\mathfrak{d})$ with dtree label $o\,'\,d$ for some $o \in O$ and some $d \in \mathbb{D}^*$. We can define *count* as a derived property as follows. Let ∇ be the sequence defined by

$$\nabla_{sym} = \nabla_{mt} = \nabla_{lam} = 0$$

$$\nabla_{app} = \lambda(\mathsf{x},\mathsf{y},\mathsf{z})(\mathsf{x}+\mathsf{y}+\mathsf{z})$$

$$\nabla_{if} = \nabla_{cart} = \lambda(\mathsf{x},\mathsf{y})(\mathsf{x}+\mathsf{y})$$

$$\nabla_{fst} = \nabla_{rst} = \lambda(\mathsf{x})\mathsf{x}$$

$$\nabla_{o} = \begin{cases} 1 & \text{if } o \in O \\ 0 & \text{otherwise} \end{cases}$$

Then $count(\mathfrak{d}, O) = (Ct(\mathfrak{d}))^{\nabla}$ for defined dtrees $\mathfrak{d}$.

Example 2: Counting symbol references. Let ∇ be the sequence defined by

$$\nabla_{sym} = 1$$

$$\nabla_{mt} = \nabla_{lam} = 0$$

$$\nabla_{app} = \lambda(\mathsf{x},\mathsf{y},\mathsf{z})(\mathsf{x}+\mathsf{y}+\mathsf{z})$$

$$\nabla_{if} = \nabla_{cart} = \lambda(\mathsf{x},\mathsf{y})(\mathsf{x}+\mathsf{y})$$

$$\nabla_{fst} = \nabla_{rst} = \lambda(\mathsf{x})\mathsf{x}$$

$$\nabla_{o} = 0$$

If $\Downarrow\mathfrak{d}$ then $(Ct(\mathfrak{d}))^{\nabla}$ is the number of symbol (value) nodes in $Ct(\mathfrak{d})$. For example, for the computation tree $Ct(\mathfrak{d}_{\mathrm{if}})$ shown in §IV.2 (Figure 13) we can easily compute $Ct(\mathfrak{d}_{\mathrm{if}})^{\nabla} = 3$.

Example 3: computation tree depth. Let ∇ be the sequence defined by

$$\nabla_{sym} = \nabla_{mt} = \nabla_{lam} = 1$$

$$\nabla_{app} = \lambda(\mathsf{x},\mathsf{y},\mathsf{z})(1+\mathsf{Max}[\mathsf{x},\mathsf{y},\mathsf{z}])$$

$$\nabla_{if} = \nabla_{cart} = \lambda(\mathsf{x},\mathsf{y})(1+\mathsf{Max}[\mathsf{x},\mathsf{y}])$$

$$\nabla_{fst} = \nabla_{rst} = \lambda(\mathsf{x})(1+\mathsf{x})$$

$$\nabla_{o} = 1$$

If $\Downarrow\mathfrak{d}$ then $(Ct(\mathfrak{d}))^{\nabla}$ is the depth (maximal path length) of $Ct(\mathfrak{d})$. In particular, we have $Ct(\mathfrak{d}_{\mathrm{if}})^{\nabla} = 5$.

Example 4: skeleton trace. Let ∇ be the sequence defined by

$$\nabla_{sym} = \mathsf{<"sym">}$$

$$\nabla_{mt} = \mathsf{<"mt">}$$

$$\nabla_{lam} = \mathsf{<"lam">}$$

$$\nabla_{app} = \lambda(\mathsf{x},\mathsf{y},\mathsf{z})\mathsf{ListMk}[\mathsf{"app"},\mathsf{x},\mathsf{y},\mathsf{z}]$$

$$\nabla_{if} = \lambda(\mathsf{x},\mathsf{y})\mathsf{ListMk}[\mathsf{"if"},\mathsf{x},\mathsf{y}]$$

$$\nabla_{cart} = \lambda(\mathsf{x},\mathsf{y})\mathsf{ListMk}[\mathsf{"cart"},\mathsf{x},\mathsf{y}]$$

$$\nabla_{fst} = \lambda(\mathsf{x})\mathsf{ListMk}[\mathsf{"fst"},\mathsf{x}]$$

$$\nabla_{fst} = \lambda(\mathsf{x})\mathsf{ListMk}[\mathsf{"rst"},\mathsf{x}]$$

$\nabla_{o} = <pname(o)>$ % $pname(o)$ is a string – the “print name” of the operation o

If $\Downarrow\mathfrak{d}$ then $(Ct(\mathfrak{d}))^{\nabla}$ is a skeleton trace of $Ct(\mathfrak{d})$ – a representation of the tree domain as a list structure with nodes labeled by strings indicating the type of the dtree labeling that node. For example

$$
\begin{aligned}
Ct(\mathfrak{d}_{\mathrm{if}})^{\nabla} = {} & \texttt{<"if" <"sym">} \\
& \qquad\qquad \texttt{<"cart"<"fst" <"sym">>} \\
& \qquad\qquad\qquad\qquad \texttt{<"app" <"lam">} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \texttt{<"rst" <"sym">>} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \texttt{<"lam">>>>}
\end{aligned}
$$

Note the correspondence between the tree structure of $Ct(\mathfrak{d}_{\mathrm{if}})$ (Figure 13 in §IV.2) and the list structure of $Ct(\mathfrak{d}_{\mathrm{if}})^{\nabla}$.

B.1.3. Derivation map for descriptions

We now define a derivation map † on descriptions corresponding to the computation tree derivation defined above. Derived descriptions compute the derived property and carry out the original computation as well (returning a derived value). The main work is defining the derivation on forms. The rest follows naturally. The derivation map is parameterized by a sequence of derivation symbols corresponding to the sequence of primitive derivations.

$$[\mathsf{D.sym}, \mathsf{D.mt}, \mathsf{D.lam}, \mathsf{D.if}, \mathsf{D.app}, \mathsf{D.cart}, \mathsf{D.fst}, \mathsf{D.rst}, \mathsf{D}.o \mid o \in \mathbb{O}]$$

We assume these symbols are new. Derived environments will map each derivation symbol to to the corresponding element of ∇.

▷ **Derivation for forms.** The derivation of a form $\mathfrak{f}$ (written $\mathfrak{f}^{\dagger}$) is defined by induction on the structure of forms.

$$
\begin{aligned}
s^{\dagger} &= [\mathsf{D.sym}, s] \\
\mathsf{mt}^{\dagger} &= [\mathsf{D.mt}, \mathsf{mt}] \\
(\lambda(s)\mathfrak{f})^{\dagger} &= [\mathsf{D.lam}, \lambda(s)\mathfrak{f}^{\dagger}] \\
\mathsf{if}(\mathfrak{f}_0, \mathfrak{f}_1, \mathfrak{f}_2)^{\dagger} &= \mathsf{let}\{[\mathsf{d0}, \mathsf{x0}] \leftarrow \mathfrak{f}_0{}^{\dagger}\} \\
&\qquad \mathsf{let}\{[\mathsf{d1}, \mathsf{x1}] \leftarrow \mathsf{if}(\mathsf{x0}, \mathfrak{f}_1{}^{\dagger}, \mathfrak{f}_2{}^{\dagger})\} \\
&\qquad\quad [\mathsf{D.if}(\mathsf{d0}, \mathsf{d1}), \mathsf{x1}] \\
\mathsf{app}(\mathfrak{f}_0, \mathfrak{f}_1)^{\dagger} &= \mathsf{let}\{[\mathsf{d0}, \mathsf{x0}] \leftarrow \mathfrak{f}_0{}^{\dagger}\} \\
&\qquad \mathsf{let}\{[\mathsf{d1}, \mathsf{x1}] \leftarrow \mathfrak{f}_1{}^{\dagger}\} \\
&\qquad \mathsf{let}\{[\mathsf{d2}, \mathsf{x2}] \leftarrow \mathsf{app}(\mathsf{x0}, \mathsf{x1})\} \\
&\qquad\quad [\mathsf{D.app}(\mathsf{d0}, \mathsf{d1}, \mathsf{d2}), \mathsf{x2}]
\end{aligned}
$$

$$\begin{aligned}\mathsf{cart}(\mathfrak{f}_0,\mathfrak{f}_1)^\dagger = {} & \mathsf{let}\{[\mathsf{d0},\mathsf{x0}] \leftarrowtail \mathfrak{f}_0{}^\dagger\}\\ & \quad \mathsf{let}\{[\mathsf{d1},\mathsf{x1}] \leftarrowtail \mathfrak{f}_1{}^\dagger\}\\ & \qquad [\mathsf{D.cart}(\mathsf{d0},\mathsf{d1}),\mathsf{x0},\mathsf{x1}]\end{aligned}$$

$$\mathsf{fst}(\mathfrak{f}_0)^\dagger = \mathsf{let}\{[\mathsf{d0},\mathsf{x0}] \leftarrowtail \mathfrak{f}_0{}^\dagger\}[\mathsf{D.fst}(\mathsf{d0}),\mathsf{fst}(\mathsf{x0})]$$

$$\mathsf{rst}(\mathfrak{f}_0)^\dagger = \mathsf{let}\{[\mathsf{d0},\mathsf{x0}] \leftarrowtail \mathfrak{f}_0{}^\dagger\}[\mathsf{D.rst}(\mathsf{d0}),\mathsf{rst}(\mathsf{x0})]$$

where d0, d1, d2, x0, x1, x2 are new symbols where they occur in the derived forms

▹ **Derivation for dtrees and other objects of t-*Rum*.**

$$\langle \mathfrak{f} \mid \xi\rangle^\dagger = \langle \mathfrak{f}^\dagger \mid \xi^\dagger\rangle$$

$$(o\,'\,d)^\dagger = o^\dagger\,'\,d^\dagger$$

$$d^\dagger = d$$

$$o^\dagger = \langle \lambda(\mathsf{x})[\mathsf{D}.o,\mathsf{o}(\mathsf{x})] \mid \mathsf{o} \leftarrowtail o\rangle$$

$$(\langle \lambda(s)\mathfrak{f} \mid \xi\rangle \in \mathbf{V})^\dagger = \langle \lambda(s)\mathfrak{f}^\dagger \mid \xi^\dagger\rangle$$

$$[a_1 \ldots a_{\mathrm{n}}]^\dagger = [a_1{}^\dagger \ldots a_{\mathrm{n}}{}^\dagger]$$

$$\xi^\dagger(s) = (\xi(s))^\dagger \qquad \text{% } s \text{ not a deriving symbol}$$

$$\xi^\dagger(\mathsf{D.sym}) = \nabla_{sym} \quad \ldots \quad \xi^\dagger(\mathsf{D}.o) = \nabla_o \quad \ldots$$

■ **Derivation commutes with pfn application.** $(\varphi\,'\,v)^\dagger = \varphi^\dagger\,'\,v^\dagger$

■ **Derivation theorem.**

$$\boldsymbol{\partial} \hookrightarrow v \rightarrow \boldsymbol{\partial}^\dagger \hookrightarrow [(Ct(\boldsymbol{\partial}))^\nabla, v^\dagger]$$

This is easily proved by computation induction. A stronger version of this theorem could be formulated expressing the sense in which the structure of the original computation is preserved.

■ **Derived recursion lemma.** The following fact about derivations of recursively defined pfns is useful in proving properties of such pfns. It follow directly (though somewhat tediously) from the definitions using simple let conversions and permutations.

$$\varphi = (\mathsf{Rec}(\langle\lambda(\mathsf{f})\lambda(\mathsf{x})\mathfrak{f} \mid \xi\rangle))^\dagger \rightarrow \varphi \cong \langle \lambda(\mathsf{x})\mathsf{let}\{[\mathsf{d},\mathsf{y}] \leftarrowtail \mathfrak{f}^\dagger\}[\mathsf{D.rec}(\mathsf{d}),\mathsf{y}] \mid \xi^\dagger\{\mathsf{f} \leftarrowtail \varphi\}\rangle$$

where

$$\begin{aligned}\mathsf{D.rec} \underset{\mathrm{df}}{=} \lambda(\mathsf{d})\mathsf{D.app}(&\mathsf{D.app}(\mathsf{D.sym},\mathsf{D.app}(\mathsf{D.sym},\mathsf{D.sym},\mathsf{D.lam}),\mathsf{D.lam}),\\ &\mathsf{D.sym},\\ &\mathsf{d})\end{aligned}$$

B.1.4. Applying the universal derivation to Tprod

We now show how to derive the definition of Mtprod from Tprod using the derivation map † followed by program transformations to simplify the derived pfn. We begin with a general derivation for Tprod and then consider the special case where the basic derivation operations are those of Example 1. For simplicity we extend the S-expression structure to include Car, Cdr, Atom and * as data operations.

Define TprodD to be the derivation of Tprod and FstTprodD, RstTprodD to be the 1^{st} and r^{st} projections of the derived computations.

$$\triangleright \quad \mathsf{TprodD} \overset{\mathrm{df}}{\longleftrightarrow} (\mathsf{Rec}(\lambda(\mathsf{TprodD}.\lambda(x) \\ \qquad \mathsf{if}(\mathsf{Atom}(x), x, *[\mathsf{TprodD}(\mathsf{Car}(x)), \mathsf{TprodD}(\mathsf{Cdr}(x))]))))^{\dagger}$$

$$\triangleright \quad \mathsf{FstTprodD} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(x)\mathsf{fst}(\mathsf{TprodD}(x))$$

$$\triangleright \quad \mathsf{RstTprodD} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(x)\mathsf{rst}(\mathsf{TprodD}(x))$$

■ **Derived Tprod lemma.**

$$\begin{aligned}
\mathsf{TprodD}(x) \cong\ & \mathsf{if}(\mathsf{Atom}(x), \\
& \quad [\mathsf{D.rec}(\mathsf{D.tpat}), x], \\
& \quad \mathsf{let}\{[\mathsf{da}, \mathsf{za}] \leftarrow \mathsf{TprodD}(\mathsf{Car}(x))\}\mathsf{let}\{[\mathsf{dd}, \mathsf{zd}] \leftarrow \mathsf{TprodD}(\mathsf{Cdr}(x))\} \\
& \qquad [\mathsf{D.rec}(\mathsf{D.tppr}(\mathsf{da}, \mathsf{dd})), *[\mathsf{za}, \mathsf{zd}])
\end{aligned}$$

$$\begin{aligned}
\mathsf{FstTprodD}(x) \cong\ & \mathsf{if}(\mathsf{Atom}(x), \\
& \quad \mathsf{D.rec}(\mathsf{D.tpat}), \\
& \quad \mathsf{D.rec}(\mathsf{D.tppr}(\mathsf{FstTprodD}(\mathsf{Car}(x)), \mathsf{FstTprodD}(\mathsf{Cdr}(x)))))
\end{aligned}$$

$$\begin{aligned}
\mathsf{RstTprodD}(x) \cong\ & \mathsf{if}(\mathsf{Atom}(x), \\
& \quad x, \\
& \quad *[\mathsf{RstTprodD}(\mathsf{Car}(x)), \mathsf{RstTprodD}(\mathsf{Cdr}(x))])
\end{aligned}$$

$$\mathsf{RstTprodD}(x) \cong \mathsf{Tprod}(x)$$

where

$$\mathsf{D.tpat} = \mathsf{D.if}(\mathsf{D.app}(\mathsf{D.sym}, \mathsf{D.dsym}, \mathsf{D.atom}), \mathsf{D.sym})$$

$$\mathsf{D.tppr} = \lambda(\mathsf{da}, \mathsf{dd})\mathsf{D.if}(\mathsf{D.app}(\mathsf{D.sym}, \mathsf{D.dsym}, \mathsf{D.atom}), \mathsf{D.tpad}(\mathsf{da}, \mathsf{db}))$$

$$\begin{aligned}
\mathsf{D.tpad} = \lambda(\mathsf{da}, \mathsf{dd})\mathsf{D.app}(\mathsf{D.sym}, & \\
& \mathsf{D.cart}(\mathsf{D.app}(\mathsf{D.sym}, \mathsf{D.car}, \mathsf{da}), \\
& \qquad \mathsf{D.app}(\mathsf{D.sym}, \mathsf{D.cdr}, \mathsf{dd})), \\
& \mathsf{D.}*)
\end{aligned}$$

■ **Derived Tprod corollary.** For ∇ as in Example 1 with $O = \{*\}$ and x ranging over number trees we have

$$\begin{aligned}
&\mathsf{TprodD(x)} \cong \mathsf{if(Atom(x),}\\
&\qquad\qquad \mathsf{[0,x],}\\
&\qquad\qquad \mathsf{let\{[da,za] \leftarrow TprodD(Car(x))\}let\{[dd,zd] \leftarrow TprodD(Cdr(x))\}}\\
&\qquad\qquad\quad \mathsf{[dd + da + 1, *[za,zd]])}\\
&\mathsf{FstTprodD(x)} \cong \mathsf{Mtprod(x)}
\end{aligned}$$

Proof of derived Tprod lemma. The key step for the derivation of the TprodD equation is to note that by the derived recursion lemma

$$\begin{aligned}
\mathsf{TprodD} \cong \lambda\mathsf{(x)let\{[d,y]} &\leftarrow \mathsf{if(Atom(x),x,*[TprodD(Car(x)),TprodD(Cdr(x))])^{\dagger}\}}\\
&\mathsf{[D.rec(d),y]}
\end{aligned}$$

and by the definition of the derivation map and laws for $\cong$ (especially let- conversions and permutations)

$$\begin{aligned}
&\mathsf{if(Atom(x),x,*[TprodD(Car(x)),TprodD(Cdr(x))])^{\dagger}}\\
&\quad \cong \mathsf{if(Atom(x),}\\
&\qquad\quad \mathsf{[D.tpat,x],}\\
&\qquad\quad \mathsf{let\{[da,za] \leftarrow TprodD(Car(x))\}let\{[dd,zd] \leftarrow TprodD(Cdr(x))\}}\\
&\qquad\qquad \mathsf{[D.tppr(da,dd),*[za,zd]])}
\end{aligned}$$

A useful trick when working in a fixed global environment such as the S-expression world is the following lemma.

■ **Global symbol lemma.** If G.sym is a global symbol such that $\mathsf{Gsym}^{\dagger} = \lambda\mathsf{(x)[D.Gsym(x),Gsym(x)]}$ for some deriving operation D.Gsym then

$$\mathsf{Gsym(y)^{\dagger}} \cong \mathsf{[D.app(D.sym,D.sym,D.Gsym(y)),Gsym(y)]}.$$

This and similar lemmas make the derivation of the equation for TprodD simpler.

Proof of derived Tprod corollary. To obtain the corollary we compute, using the derivation operations of Example 1, that

$$\mathsf{D.rec}(d) = d \qquad \mathsf{D.tpat} = 0 \qquad \mathsf{D.tppr}(d_{\mathrm{a}}, d_{\mathrm{d}}) = 1 + d_{\mathrm{a}} + d_{\mathrm{d}}$$

B.1.5. Possible elaborations

The computation tree derivation defined above accounts for many of the intensional properties of computations that have been discussed. There are several possible elaborations that would allow even more to be expressed.

One possibility is for primitive derivations to take the value of each subcomputation as an argument in addition to the derived property. This would allow a more complete trace to be defined which includes not only the data operation names but also the arguments. The derived program *trace.rev* (§I.3) is an example. More generally deriving functions for data operations could make the cost of the operation depend on the argument.

A further elaboration is to modify the transformation to turn each value into an object (pfn) that handles messages. For example the follow message response pairs could be included.

"val" ⇒ the transformed object

"type" ⇒ "data", "opc", or "pfn"

"name" ⇒ an identifying pname

Derivation pfns then have more information about the arguments available. For example the applications of a pfn with a given name could be counted.

B.2. Searching tree structures

Tree structures are important in practice as well as in theory. Tree-structured search spaces are one example of their use in practice. A tree-structured search space is a set P together with a function ϑ from P to P^*. P is called the search space and elements p of P are called positions. ϑ is called the successors function. ϑ determines the sequence of immediate successors of any position in the space and hence the tree of positions below any given position. (Recall that we grow our trees from the root downward.) The reason such spaces are called search spaces is that they are used to organize information to enable a program to search for solutions to a problem. For example P could be the set of legal board positions in a game such as chess or checkers. The successors of a position are given by the legal moves from that position. One is searching for sequences of moves leading to a winning position. Another example is pattern matching. Here P is a set of triples (π, ι, μ) where π is a pattern (containing some pattern variables), ι is the object to be matched and μ is a partial match, assigning values to some of the pattern variables. The successors are triples (π, ι, μ') where μ' is an extension of μ.

In the following we assume that the search space P is a subset of the computation domain and that the successors function ϑ is a pfn or data operation. A search strategy is an ordering on the nodes of a search space – the order in which nodes are to be visited in the search process. A *search pfnl* for a particular search strategy is a pfnl that maps a tree (i.e. a successor and a root position) to a stream of positions generated in the order prescribed by the strategy. Our idea is to define search pfnls, prove general properties of the search pfnls and use this basis to define particular search pfns and prove properties of these pfns. Here we shall consider a particularly simple strategy called ***depth-first search***. The ideas generalize to alternative and more complex strategies such as those described in Burstall [1968]. A depth-first search of a tree goes down the left-most branch, visiting each node on the way until it reaches a leaf. Then it *backtracks* to the nearest previous node having an unvisited successor and does a depth-first search from the leftmost unvisited successor.

We begin by developing enough of the mathematics of tree-structured spaces and depth-first search to provide a semantic basis for expressing and proving properties of search pfnls. The tree structure given by a successor function and a position and additional related concepts are defined. A depth-first enumeration function is defined, to make precise the notion of depth-first order, and key properties of this function are given. Finally we define a depth-first search pfnl and prove some basic properties. Two applications are given: generating the fringe of an S-expression and a simple pattern matcher. Definitions of operations on S-expressions can be found in §IV.5 and definitions and properties related to streams can be found in §IV.6.

B.2.1. Position trees

Let $P \subset \mathbf{V}$ be a set of positions and fix $\vartheta : [P \nrightarrow P^*]$ a successors operation for P. $p, p_0, \ldots$ will range over P.

▷ **The tree given by (ϑ, p).** $Tree(\vartheta, p)$, the tree given by (ϑ, p), is the least tree $<\Delta, \Lambda> \in \mathbb{T} : <P>$ such that

$$\begin{array}{lll} (\Delta) & \square \in \Delta & \nu \in \Delta \wedge \Lambda(\nu) = p_0 \wedge j < |\vartheta(p_0)| \rightarrow [\nu, j] \in \Delta \\ (\Lambda) & \Lambda(\square) = p & \Lambda([\nu, j]) = \vartheta(p_0)\downarrow_j \end{array}$$

Note that $[\nu, j+1] \in \Delta \rightarrow [\nu, j] \in \Delta$. In this section we will only consider tree domains $\Delta \in \mathbb{T}$ that have this property. (The *no-gaps* assumption.)

▷ **Degree of a node in a tree domain.** We also assume that each node ν of Δ has a finite number of immediate successors. This number is called the degree of ν in Δ (written $deg(\Delta, \nu)$).

$$deg(\Delta, \nu) \underset{\mathrm{df}}{=} |\{j \mid [\nu, j] \in \Delta\}|$$

Note that by the no-gaps assumption on tree domains,

$$deg(\Delta,\nu) \underset{\text{df}}{=} \mu(n)([\nu,j] \notin \Delta) \quad \text{and} \quad \nu \in \Delta \wedge j < deg(\Delta,\nu) \rightarrow [\nu,j] \in \Delta.$$

The sequence of immediate successors of ν in Δ is the sequence of nodes $[\nu,j]$ in Δ. Since nodes themselves are sequences of numbers it is necessary to package the nodes as elements in order to form the successor sequence. The simplest form of package is $\mathsf{K}(\nu)$. (Recall $\mathsf{K} = \lambda(\mathsf{x},\mathsf{y})\mathsf{x}$.) For readability we define pfns Close and Open which package and unpackage sequences.

▷ $\mathsf{Close} \overset{\text{df}}{\longleftrightarrow} \mathsf{K}$

▷ $\mathsf{Open} \overset{\text{df}}{\longleftrightarrow} \lambda(\mathsf{s})\mathsf{s}(\mathsf{mt})$

▷ **Immediate successor sequence.** The immediate successor sequence $isuc(\Delta,\nu)$ for $\nu \in \Delta$ is defined by

$$|isuc(\Delta,\nu)| = deg(\Delta,\nu) \quad \text{and} \quad i < deg(\Delta,\nu) \rightarrow isuc(\Delta,\nu)\downarrow_i = \mathsf{Close}([\nu,i])$$

Note that for $Tree(\vartheta,p) = \langle\Delta,\Lambda\rangle$ and $\nu \in \Delta$

$$\vartheta(\Lambda(\nu)) = \mathsf{Collect}(\Lambda \circ \mathsf{Open}, isuc(\Delta,\nu))$$

Two additional notions that are useful for talking about enumeration of nodes in a tree domain are the *above* and *left-of* relations on nodes.

▷ **Above relation.** A node ν_0 is above another node ν_1 (written $\nu_0 \leq_{above} \nu_1$) if ν_0 is on the path to ν_1.

$$\nu_0 \leq_{above} \nu_1 \underset{\text{df}}{\equiv} (\exists \nu_2)(\nu_1 = [\nu_0,\nu_2])$$

▷ **Left-of relation.** The node ν_0 is to the left of the node ν_1 (written $\nu_0 <_{left} \nu_1$) is defined by

$$\nu_0 <_{left} \nu_1 \underset{\text{df}}{\equiv} (\exists i < j)(\exists \nu)([\nu,i] \leq_{above} \nu_0 \wedge [\nu,j] \leq_{above} \nu_1)$$

■ **Independence.** If ν_0 is to the left of ν_1 the tree below ν_0 is disjoint from the tree below ν_1.

$$\nu_0 <_{left} \nu_1 \rightarrow \neg(\nu_0 \leq_{above} \nu \wedge \nu_1 \leq_{above} \nu)$$

B.2.2. Depth-first enumeration

The function $df(\Delta, n)$ enumerates nodes in a tree domain in depth-first order. $\Box$ is the least node (the 0-th stage). At each stage, the next node is the leftmost successor in Δ of the current node unless the current node has no successors. If the current node has no successors, the next node is the left most unvisited successor of the nearest ancestor with an unvisited successor. This is found by the backtracking function *back*. [1] At each stage of enumeration, the remainder of the tree domain $df^{>}(\Delta, n)$ is a forest (a sequence of subtrees). The sequence of roots of this forest corresponds to the stack used in backtracking in a depth-first search. The spanning function $span(\Delta, n)$ gives the sequence of roots (the span) of the remainder at the n-th stage, leftmost first. Thus every node in $df^{>}(\Delta, n)$ is below a unique node of $stack(\Delta, n)$ and every node of Δ below a node of $stack(\Delta, n)$ is in $df^{>}(\Delta, n)$.

▹ **Depthfirst enumeration.** $df(\Delta, n)$ is defined by induction on n. $back(\Delta, \nu)$ is defined by induction on ν.

$$back(\Delta, \Box) = \Box \quad \text{and} \quad back(\Delta, [\nu, j]) = \begin{cases} [\nu, j+1] & \text{if } [\nu, j+1] \in \Delta \\ back(\Delta, \nu) & \text{otherwise} \end{cases}$$

$$df(\Delta, 0) = \Box$$

$$df(\Delta, n) = \nu \wedge [\nu, 0] \in \Delta \rightarrow df(\Delta, n+1) = [\nu, 0]$$

$$df(\Delta, n) = \nu \wedge [\nu, 0] \notin \Delta \rightarrow df(\Delta, n+1) = back(\Delta, \nu)$$

▹ **Depth-first remainder.** $df^{>}(\Delta, n) \underset{\text{df}}{=} \Delta - \{df(\Delta, j) \mid j \leq n\}$

▹ **Spanning function.** $span(\Delta, n)$ is defined by induction on n.

$$span(\Delta, 0) = isuc(\Delta, \Box)$$

$$span(\Delta, n) = [\mathsf{Close}(\nu), v] \rightarrow span(\Delta, n+1) = [isuc(\Delta, \nu), v]$$

$$span(\Delta, n) = \Box \rightarrow span(\Delta, n+1) = \Box$$

▪ **Backtracking lemma.** The key properties of the spanning and backtracking functions are (left) and (stack). Proof is by induction on n.

(left) $\quad i < j < |span(\Delta, n)| \rightarrow \mathsf{Open}(span(\Delta, n)\downarrow_i) <_{left} \mathsf{Open}(span(\Delta, n)\downarrow_j)$

(stack) $\quad span(\Delta, n)\downarrow_j = \mathsf{Close}(\nu) \wedge back(\Delta, \nu) \neq \Box \rightarrow$

$$span(\Delta, n)\downarrow_{j+1} = \mathsf{Close}(back(\Delta, \nu))$$

[1] By the usual definition, ν_0 is before ν_1 in the depth-first order iff ν_0 is above (and not equal to) or to the left of ν_1. On a tree domain depth-first enumeration agrees with depth-first order on any finite initial segment.

■ **Depth-first enumeration theorem.** The key facts about *df* and *span* are that the nodes $df(\Delta, j)$ for $j < |\Delta|$ are distinct elements of Δ and for $j+1 < |\Delta|$ the first element of spanning sequence of the j-th remainder is the next element of the depth-first enumeration.

(df.enum) $j < |\Delta| \rightarrow df(\Delta, j) \in \Delta \wedge i < j < |\Delta| \rightarrow df(\Delta, i) \neq df(\Delta, j)$

(df.next) $j+1 < |\Delta| \rightarrow 1^{st}(span(\Delta, j)) = \mathsf{Close}(df(\Delta, j+1))$

The depth-first enumeration theorem is an immediate consequence of the following lemma.

■ **Characterization of depth-first enumeration and remainder.** (card) says that the remainder of Δ is reduced by one at each step of the enumeration. (down) says that the remainder at each stage downward closed and thus is a union of disjoint subtrees. (span) is the characterizon of spanning given above.

(card) $n < |\Delta| \rightarrow |df^{>}(\Delta, n)| = |\Delta| - n - 1$

(down) $n < |\Delta| \wedge \nu \in df^{>}(\Delta, n) \wedge [\nu, j] \in \Delta \rightarrow [\nu, j] \in df^{>}(\Delta, n)$

(span) $\mathsf{Close}(\nu) \in span(\Delta, n) \rightarrow \nu \in df^{>}(\Delta, n) \wedge$

$n < |\Delta| \wedge \nu \in df^{>}(\Delta, n) \rightarrow (\exists \nu_0)\mathsf{Close}(\nu_0) \in span(\Delta, n) \wedge \nu_0 \leq_{above} \nu$

(pop) $n+1 < |\Delta| \rightarrow 1^{st}(span(\Delta, n)) = \mathsf{Close}(df(\Delta, n+1))$

These facts are easily proved by induction on n using the backtracking lemma.

B.2.3. Depth-first search pfnl

Dfst is a pfnl that takes a tree (ϑ, p) and returns a stream that generates positions labeling nodes of the tree in depth-first order. More generally, Dfst takes a successor operation s and a sequence of positions [p, z] labeling the roots of a sequence of trees and returns a stream generating positions labeling the nodes of the sequence of trees.

▷ $\mathsf{Dfst} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{s})\mathsf{Rec}(\lambda(\mathsf{df})\lambda[\mathsf{p},\mathsf{z}]\lambda()\mathsf{ifmt}(\mathsf{p},\mathsf{mt},[\mathsf{p},\mathsf{df}(\mathsf{s}(\mathsf{p}),\mathsf{z})]))$

■ **Dfst enumerates depthfirst.** For $Tree(\vartheta, p) = \langle\Delta, \Lambda\rangle$ and $n < |\Delta|$

$$\mathsf{Dfst}(\vartheta, p)^{(n)} = \Lambda(df(\Delta, n))$$

$$\mathsf{Dfst}(\vartheta, p)^{>n} = \mathsf{Dfst}(\vartheta, \mathsf{Collect}(\Lambda \circ \mathsf{Open}, span(\Delta, n)))$$

Proof is by induction on n using the depth-first enumeration theorem.

■ Dfst **corollary.** If $|\Delta| < \omega$ then $\mathsf{Filter}(\varphi, \mathsf{Dfst}(\vartheta, p))$ generates

$$\{p_0 \mid (\exists\nu \in \Delta)(\Lambda(\nu) = p_0) \wedge \varphi(p_0) \neq \Box\}$$

in depth-first order.

■ **Dfst enumeration of forests.** A further useful fact about the depth-first enumeration stream is that the depth-first stream for a forest is the concatenation of streams for individual trees. If $\mathit{Tree}(\vartheta, p_i) = {<}\Delta_i, \Lambda_i{>}$ for $0 \leq i \leq k$ then

$$\mathsf{Dfst}(\vartheta, [p_0 \ldots p_k]) \overset{s}{=} \mathsf{Dfst}(\vartheta, p_0) \diamond \ldots \diamond \mathsf{Dfst}(\vartheta, p_k)$$

Proof is by induction on k using the Dfst forest lemma and properties of stream concatenation.

■ **Dfst forest lemma.** For $\mathit{Tree}(\vartheta, p) = {<}\Delta, \Lambda{>}$ and $n < |\Delta|$

$$\mathsf{Dfst}(\vartheta, [p, v])^{(n)} = \mathsf{Dfst}(\vartheta, p)^{(n)}$$
$$\mathsf{Dfst}(\vartheta, [p, v])^{>n} = \mathsf{Dfst}(\vartheta, [\mathsf{Collect}(\Lambda \circ \mathsf{Open}, \mathit{span}(\Delta, n)), v])$$

Proof is by induction on n.

B.2.4. The fringe of an S-expression

As our first application of searching, we let $P = \mathbb{D}_{\mathrm{sexp}}$ with successor operation PairUn and show how the fringe of an S-expression can be generated by searching the corresponding tree for atoms. The fringe of an S-expression a is the sequence of atoms occurring in a in left to right order. The pfn Fr computes the fringe (as sequence of atoms) in the standard way by concatenating the fringe of the components until a non-pair is reached. FrS returns a stream generating elements of the fringe by searching for atoms in depth first order.

▷ $\mathsf{Fr} \overset{\mathrm{df}}{\longleftrightarrow} \mathsf{Rec}(\lambda(\mathsf{Fr})\lambda(\mathsf{x})\mathsf{if}(\mathsf{Atom}(\mathsf{x}), \mathsf{x}, [\mathsf{Fr}(\mathsf{Car}(\mathsf{x})), \mathsf{Fr}(\mathsf{Cdr}(\mathsf{x}))]))$

▷ $\mathsf{FrS} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{x})\mathsf{Filter}(\mathsf{Atom}, \mathsf{Dfst}(\mathsf{PairUn}, \mathsf{x}))$

■ **Fringe theorem.** For S-expressions a, $\mathsf{Fr}(a)$ and FrS have the same length and corresponding elements are equal.

$$|\mathsf{Fr}(a)| = |\mathsf{FrS}(a)|_s \quad \text{and} \quad j < |\mathsf{Fr}(a)| \rightarrow \mathsf{Fr}(a){\downarrow}_j = \mathsf{FrS}(a)^{(j)}$$

This follows easily from the fringe lemma.

■ **Fringe lemma.** For S-expressions a Streamify composed with Fr gives the same stream as FrS.

$$\mathsf{Streamify}(\mathsf{Fr}(a)) \overset{s}{\doteq} \mathsf{FrS}(a)$$

Proof of fringe lemma: by S-expression induction using properties of concatenation for Streamify, Filter and Dfst. Note that for $Tree(\mathsf{PairUn}, a) = \langle\Delta, \Lambda\rangle$ $|\Delta|$ is finite. The key points for the proof are

$$\mathsf{Atom}(a) \neq \square \rightarrow \mathsf{Streamify}(\mathsf{Fr}(a)) = \mathsf{Streamify}(a) \overset{s}{\doteq} \lambda()[a, \mathsf{MtStream}] \overset{s}{\doteq} \mathsf{FrS}(a)$$

$$\mathsf{Atom}(a) = \square \rightarrow$$

$$\mathsf{Streamify}(\mathsf{Fr}(a)) \overset{s}{\doteq} \mathsf{Streamify}(\mathsf{Fr}(\mathsf{Car}(a))) \diamond \mathsf{Streamify}(\mathsf{Fr}(\mathsf{Cdr}(a)))$$

$$\mathsf{Atom}(a) = \square \rightarrow \mathsf{FrS}(a) \overset{s}{\doteq} \mathsf{Filter}(\mathsf{Atom}, \mathsf{Dfst}(\mathsf{PairUn}, [\mathsf{Car}(a), \mathsf{Cdr}(a)]))$$

$$\overset{s}{\doteq} \mathsf{Filter}(\mathsf{Atom}, \mathsf{Dfst}(\mathsf{PairUn}, \mathsf{Car}(a)) \diamond \mathsf{Dfst}(\mathsf{PairUn}, \mathsf{Cdr}(a)))$$

$$\overset{s}{\doteq} \mathsf{Filter}(\mathsf{Atom}, \mathsf{Dfst}(\mathsf{PairUn}, \mathsf{Car}(a))) \diamond \mathsf{Filter}(\mathsf{Atom}, \mathsf{Dfst}(\mathsf{PairUn}, \mathsf{Cdr}(a)))$$

B.2.5. Pattern matching using tree searching pfnls

In order to illustrate the main points as simply as possible we consider the simplest notion of pattern where multiple matches are possible. A pattern π is a list of pattern variables, an instance ι is a list of integers, and a match μ is finite map associating sequences of constants to pattern variables. The instantiation of a pattern π by a match μ is the list made from the sequence obtained by collecting the sequences of constants associated to the pattern variables of π by μ. We say μ matches π to ι if the domain of μ contains the set of pattern variables of π and ι is the instantiation of π by μ. As an example let A and B be pattern variables, $\pi = \langle \mathsf{A}, \mathsf{B}, \mathsf{A}\rangle$ and $\iota = \langle 0, 0\rangle$. Then $\{\mathsf{A} \leftarrow 0, \mathsf{B} \leftarrow \square\}$ and $\{\mathsf{A} \leftarrow \square, \mathsf{B} \leftarrow [0,0]\}$ are both matches of π to ι.

To represent this in the S-expression world we let pattern variables be strings. *Pat* is the set of patterns and *Inst* is the set of pattern instances.

$$Pat = \mathsf{ListMk}(\mathbb{D}^*_{\mathrm{str}}) \quad \text{and} \quad Inst = \mathsf{ListMk}(\mathbb{D}^*_{\mathrm{int}})$$

π, π_0, ...will range over *Pat* and ι, ι_0, ...will range over *Inst*.

▷ **The set of matches.** The set of pattern variables $Set(\pi)$ in a pattern π is defined by

$$Set(\pi) \underset{\mathrm{df}}{=}= \{a \in \mathsf{ListUn}(\pi)\}$$

We use pfns to represent finite maps corresponding to matches. MtMat is the empty match and $\mathsf{Bind}(\mu, z, v)$ adds the binding $z \leftarrow v$ to the match μ.

▷ $\mathsf{MtMat} \overset{\mathrm{df}}{\hookleftarrow} \lambda(\mathsf{z})\mathsf{z}$

▷ $\mathsf{Bind} \overset{\mathrm{df}}{\hookleftarrow} \lambda(\mathsf{m},\mathsf{z},\mathsf{v})\lambda(\mathsf{x})\mathsf{if}(\mathsf{StrEq}(\mathsf{z},\mathsf{x}),\mathsf{v},\mathsf{m}(\mathsf{x}))$

Mat, the set of (partial) matches, is generated from the empty match by the binding operation. Thus *Mat* is the least set of pfns satisfying

$$Mat = \mathsf{MtMat} \oplus \mathsf{Bind}(Mat, \mathbb{D}_{\mathrm{str}}, \mathbb{D}^*_{\mathrm{int}})$$

μ, μ_0, ...will range over *Mat*.

■ **The domain of a match pfn.** The choice of the identity pfn as the empty match means we can characterize the domain of a match pfn as the set of strings for which the match pfn returns a non-string.

$$dom(\mu) = \{z \in \mathbb{D}_{\mathrm{str}} \mid \mu(z) \notin \mathbb{D}_{\mathrm{str}}\}$$

This is convenient for computing extensions.

▷ **Equality of match pfns.** μ is the same match as μ' (written $\mu \sim \mu'$) if they correspond to the same set of bindings.

$$\mu \sim \mu' \underset{\mathrm{df}}{\equiv} dom(\mu) = dom(\mu') \wedge (\forall z \in dom(\mu))\mu(z) = \mu'(z)$$

Pattern instantiation is described by Pinst. This corresponds directly to the description given above.

▷ $\mathsf{Pinst} \overset{\mathrm{df}}{\hookleftarrow} \lambda(\mathsf{p},\mathsf{m})\mathsf{ListMk}(\mathsf{Collect}(\mathsf{m},\mathsf{ListUn}(\mathsf{p})))$

▷ **The set of matches.** The set of matches $Matches(\pi, \iota)$ of a pattern π to an instance ι is defined by

$$Matches(\pi, \iota) \underset{\mathrm{df}}{=} \{\mu \in Mat \mid dom(\mu) = Set(\pi) \wedge \mathsf{Pinst}(\pi, \mu) = \iota\}$$

We have restricted the set of matches of π to ι to maps whose domain is the set of pattern variables in π in order to have a finite set of matches.

To search for all matches of a pattern π to an instance ι one begins with the empty match and tries extensions binding the next unbound pattern variable in π to initial segments of the unmatched portion of ι. *Pos*, the domain of positions in the pattern search space is the set of triples (closed sequences of length three) from $[Pat \times Inst \times Mat]$.

$$Pos = \mathsf{Close}[Pat, Inst, Mat]$$

The successors of a position are determined as follows. A position (π, ι, μ) has no successors if π is the empty list. Suppose $\pi = z \bullet \pi_1$ and z is in the domain of μ. If $\iota = \mathsf{ListExtend}(\mu(z), \iota_1)$ for some ι_1 then there is one successor (π_1, ι_1, μ). If $\mu(z)$ in not an initial segment of ι then there are no successors, since μ has no extension matching π to ι. Suppose $\pi = z \bullet \pi_1$ and z is not in the domain of μ. Then the successors of (π, ι, μ) are triples (π_1, ι_1, μ_1) where $\iota = \mathsf{ListExtend}(v, \iota_1)$ and $\mu_1 = \mathsf{Bind}(\mu, z, v)$. Successors of a position are computed by Psuc.

```
▷    Psuc ⟷df λ(pos)let{[p,e,m] ← Open(pos)}
                          if(MtlP(p),
                             mt,
                             let{[z,p] ← PairUn(p)}let{v ← m(z)}
                                if(StrP(p),
                                   Pss(z,p,m,mt,e),
                                   let{e1 ← ListAfter(v,e)}
                                      if(e1,Close[p,e1,m],mt)))

▷    Pss ⟷df λ(z,p,m)Rec(λ(ps)λ(v,e)
                            [Close[p,e,Bind(m,z,v)],
                            if(MtlP(e),mt,Pss(z,p,m,[v,Car(e)],Cdr(e)))]
```

A position is a match if the pattern and instance components are both empty lists. MatchP tests for matches. Thus matches of π to ι are generated by searching the tree of positions below $(\pi, \iota, \mathsf{MtMat})$ and filtering out non-matches. This is described by MatchS.

```
▷    MatchP ⟷df λ(pos)let{[p,e,m] ← Open(pos)}and(MtlP(p),MtlP(e))

▷    MatchS ⟷df λ(p,e)Filter(MatchP,Dfst(Psuc,Close[p,e,Mtmat]))
```

■ **MatchS theorem.** $\mathsf{MatchS}(\pi, \iota)$ generates $Matches(\pi, \iota)$.

$$\mu \in Matches(\pi, \iota) \;\leftrightarrow\; (\exists n)(\mathsf{MatchS}(\pi, \iota)^{(n)} \sim \mu)$$

Proof of MatchS theorem. Fix a pattern π and an instance ι and let

$$Tree(\mathsf{Psuc})(\mathsf{Close}[\pi,\iota,\mathsf{MtMat}]) = \langle\Delta,\Lambda\rangle$$

Using properties of Dfst we need only show

(i) $|\Delta| < \omega$

(ii) $Matches(\pi,\iota) = \{\mu \in Mat \mid (\exists\nu \in \Delta, \mu' \sim \mu)(\mathsf{Open}(\Lambda(\nu)) = [\mathsf{Nil},\mathsf{Nil},\mu'])\}$

(i) and (ii) follow from Lemmas 1, 2.

■ **Lemma 1.**

$$\begin{aligned}&\nu \in \Delta \land \mathsf{Open}(\Lambda(\nu)) = [\pi_1,\iota_1,\mu] \rightarrow\\ &\quad(\exists\pi_0,\iota_0)(\pi = \pi_0 \diamond \pi_1 \land \iota = \iota_0 \diamond \iota_1 \land\\ &\qquad\quad dom(\mu) = Set(\pi_0) \land |\nu| = |\pi_0| \land \mathsf{Pinst}(\pi_0,\mu) = \iota_0)\end{aligned}$$

Proof by induction on $\nu \in \Delta$

■ **Lemma 2.**

$$\begin{aligned}&\mu \in Matches(\pi,\iota) \land \pi = \pi_0 \diamond \pi_1 \rightarrow\\ &\quad(\exists\iota_0,\iota_1,\nu \in \Delta, \mu_0 \in Mat)\\ &\qquad(\mathsf{Pinst}(\pi_0,\mu) = \iota_0 \land \iota = \iota_0 \diamond \iota_1 \land dom(\mu_0) = Set(\pi_0) \land\\ &\qquad\ \mathsf{Open}(\Lambda(\nu)) = (\pi_1,\iota_1,\mu_0) \land (\forall z \in dom(\mu_0))(\mu(z) = \mu_0(z)))\end{aligned}$$

Proof: by induction on π_0

To complete the proof of the MatchS theorem, note that by Lemma 1 we have

$$|\Delta| < \omega \quad \text{and} \quad \nu \in \Delta \land \mathsf{Open}(\Lambda(\nu)) = [\mathsf{Nil},\mathsf{Nil},\mu] \rightarrow \mu \in Matches(\pi,\iota)$$

and by Lemma 2 (taking $\pi_0 = \pi$) we have

$$\mu \in Matches(\pi,\iota) \rightarrow (\exists\nu \in \Delta, \mu' \sim \mu)(\mathsf{Open}(\Lambda(\nu)) = [\mathsf{Nil},\mathsf{Nil},\mu'])$$

$\square_{MatchS}$

B.2.6. Trees given by successor streams

A variant on the notion of tree-structured space is to replace successor sequence by successor stream. This could be useful if the cost of generating successors is large and only a few will generally be needed. It also allows for the possibility of infinitely branching trees.

Dfirst is a search pfnl for depth-first search using successor streams. It calls Dfstr with the successor stream, a singleton stream containing the root position, and the empty stream. Dfstr has a successor pfn parameter sc and uses a current successor stream s and a continuation stream c which serves as a stack of successor streams. When s is empty, the continuation stream is used to generate the next element. Otherwise the first element p of s is returned as the first element of the stream. The rest of the stream has successor stream the successors of p and continuation stream the stream with the rest of s as successor stream and c as continuation stream.

$$\triangleright \quad \mathsf{Dfirst} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{sc},\mathsf{pos})\mathsf{Dfstr}(\mathsf{sc},\lambda()[\mathsf{pos},\mathsf{Mtstream}],\mathsf{Mtstream})$$

$$\triangleright \quad \mathsf{Dfstr} \overset{\mathrm{df}}{\longleftrightarrow} \lambda(\mathsf{sc})\mathsf{Rec}(\lambda(\mathsf{dfs})\lambda(\mathsf{s},\mathsf{c})\lambda()\mathsf{let}\{[\mathsf{p},\mathsf{ss}] \leftarrow \mathsf{s}[]\}$$
$$\mathsf{ifmt}(\mathsf{p},\mathsf{c}[],[\mathsf{p},\mathsf{dfs}(\mathsf{sc}(\mathsf{p}),\mathsf{dfs}(\mathsf{ss},\mathsf{c}))]))$$

Exercise 1. Prove that

$$(\forall p \in P)(\mathsf{Streamify}(\vartheta_0(p)) \overset{s}{=} \vartheta_1(p)) \rightarrow \mathsf{Dfst}(\vartheta_0,p) \overset{s}{=} \mathsf{Dfirst}(\vartheta_1,p)$$

Bibliography

Abelson, H. and G. J. Sussman

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

Aczel, P.

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

Backus, J.

[1978] Can programming be liberated from the von Neumman style? A functional style and its algebra of programs, *Comm. ACM*, **21**, pp. 613–641.

Barendregt, H.

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

Barwise, J., editor

[1977] *Handbook of mathematical logic* (North-Holland, Amsterdam).

Bobrow, D. G. and B. Wegbreit

[1973] A model and stack implementation of multiple environments, *Comm. ACM*, **16**, pp. 591–603.

Boyer, R. S. and J. S. Moore

[1979] *A computational logic* (Academic Press, New York).

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.

[1981] *Iswim programming manual*, IBM Research Report, RA 129.

Burstall, R. M.

[1968] Writing search algorithms in functional form, in: *Machine intelligence 3*, edited by D. Michie, (Edinburgh University Press), pp. 373–385.

[1969] Proving properties of programs by structural induction, *The Computer Journal*, **12**, pp. 41–48.

Burstall, R. M. and J. Darlington

[1976] A system which automatically improves programs, *Acta Informatica*, **6**, pp. 41–60.

[1977] A transformation system for developing recursive programs, *J. ACM*, **24**, pp. 44–67.

Burstall, R. M. and R. J. Popplestone

[1968] POP-2 reference manual, in: *Machine intelligence 2*, edited by E. Dale and D. Michie, (American Elsevier, New York) pp. 205–403.

Cartwright, R. and J. McCarthy

[1979] Recursive programs as functions in a first order theory, in: *Proceedings of the international conference on mathematical studies of information processing, Kyoto, Japan.*

Church, A.

[1941] *The calculi of lambda-conversion*, Annals of mathematics studies, vol. 6 (Princeton University Press).

Clocksin, W. F. and C. S. Mellish

[1984] *Programming in Prolog*, second edition, (Springer, Berlin).

Conway, M.

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

deBruijn, N. G.

[1972] Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the Church-Rosser theorem. *Indag. Math.*, **34**, pp. 381–392.

Feferman, S.

[1975] Non-extensional type-free theories of partial operations and classifications, I. in: *Proof theory symposium, Kiel 1974*, edited by J. Diller and G. H. Müller, Lecture notes in mathematics, no. 500 (Springer, Berlin) pp. 73–118.

[1977] Inductive schemata and recursively continuous functionals, in: *Logic colloquium 76*, edited by R. O. Gandy and J. M. E. Hyland (North-Holland, Amsterdam) pp.373–392.

[1979] Constructive theories of functions and classes, in: *Logic colloquium 78* edited by M. Boffa, D. van Dalen and K. McAloon (North-Holland, Amsterdam) pp. 159–224.

[1981] A theory of variable types, (to appear in *Proceedings VIth Latin American logic symposium*, Bogota, Colombia).

[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.

Fischer, M. J.

[1972] Lambda calculus schemata, in: *Proceedings of an ACM conference on proving assertions about programs*, pp. 104–109.

Fenstad, J. E.

[1980] *General recursion theory: an axiomatic approach* (Springer, Berlin).

Friedman, D. P. and M. Wand

[1984] Reification: reflection without metaphysics, in: *Proceedings of the 1984 ACM symposium on Lisp and functional programming*, pp. 348–355.

Friedman, D. P. et.al.

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

Goad, C.

[1980] *Computational uses of the manipulation of formal proofs*, Ph.D. thesis, Stanford University.

Gödel, K.

[1934] On undecidable propositions of formal mathematical systems, reprinted in: M. Davis, *The undecidable, basic papers on undecidable propositions, unsolvable problems and computable functions* (Raven Press, Hewlett, N.Y.; 1965) pp. 39–74.

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).

Goldberg, A. and D. Robson

[1983] *Smalltalk-80. The language and its implementation* (Addison-Wesley, Reading).

Gordon, M. J. C.

[1973] An investigation of *lit*: where
$lit((A_1,\ldots,A_n),A_{n+1},f) = f(A_1,f(A_2,\ldots,f(A_n,A_n+1),\ldots))$,
Memorandum MIP-R-101, School of Artificial Intelligence, University of Edinburgh.

[1975] Operational reasoning and denotational semantics, Stanford Artificial Intelligence Laboratory Memo AIM-264, Computer Science Department, Stanford Univiserity.

Gordon, M. J., A. J. Milner and C. P. Wadsworth

[1979] *Edinburgh LCF: A mechanized logic of computation*, Lecture notes in computer science, no. 78, (Springer, Berlin).

Grzegorczyk, A., A. Mostowski and C. Ryll-Nardzewski

[1958] The classical and the ω-complete arithmetic, *J. Symbolic Logic*, **23**, pp. 188–206.

Hewitt, C.

[1977] Viewing control structures as patterns of passing messages, *Artificial Intelligence*, **8**, pp. 323–363.

Kechris, A. S. and Y. N. Moschovakis

[1977] Recursion in higher types, in: *Barwise 1977*, pp. 682–737.

Ketonen, J.

[1984] EKL – a mathematically oriented proof checker, in: *Seventh International Conference on Automated Deduction, Napa, California*, edited by R. E. Shostak, Lecture notes in computer science, no. 170 (Springer, Berlin).

Kleene, S. C.

[1936a] λ-definablility and recursiveness, *Duke Mathematical Journal*, **2**, pp. 340–353.

[1936b] General recursive functions of natural numbers, *Math. Ann.*, **112**, pp. 727–742.

[1952] *Introduction to metamathematics*, (North-Holland, Amsterdam).

[1959] Recursive functionals and quantifiers of finite types I, *Trans. Am. Math. Soc.*, **91**, pp. 1–52.

[1963] Recursive functionals and quantifiers of finite types II, *Trans. Am. Math. Soc.*, **108**, pp. 106–142.

[1979] Origins of recursive function theory, in: *Proceedings of the 20th symposium on the foundations of computer science*, pp. 371–382.

Landin, P. J.

[1964] The mechanical evaluation of expressions, *Computer Journal*, **6**, pp. 308–320.

[1965] A correspondence between Algol60 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.

Lloyd, J. W.

[1984] *The foundations of logic programming*, (Springer, Berlin).

Manna, Z.

[1969] The correctness of programs, *J. Computer and System Sciences*, **3**, pp. 119–127.

[1978] *Six lectures on the logic of computer programming*, SIAM CBMS-NSF regional conference series in applied mathematics, no. 31.

Manna, Z. and R. Waldinger

[1980] A deductive approach to program synthesis, *ACM transactions on programming languages and systems*, **2**, pp. 92–121.

McCarthy, J.

[1960] Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM*, **3**, pp. 184–195.

[1963a] Towards a Mathematical Science of Computation, in: *Information processing: Proceedings of the IFIP congress 62*, edited by C. M. Popplewell (North-Holland, Amsterdam) pp. 21-28.

[1963] A basis for a mathematical theory of computation, in: *Computer programming and formal systems*, edited by P. Braffort and D. Herschberg (North-Holland, Amsterdam) pp. 33–70.

[1978] History of Lisp, in: *Proceedings of the ACM conference on history of programming languages, ACM SIGPLAN Notices, bf 13*; reprinted in: *History of programming languages*, edited by R. L. Wexelblat (Academic Press, New York; 1981) pp. 173–197.

McCarthy, J. et. al.

[1962] *Lisp 1.5 programmer's manual* (M. I. T. Press).

McCarthy, J. and C. Talcott

[1980] *Lisp: programing and proving*, Course notes, Computer Science Department, Stanford University (under revision for publication as a book).

Milne, R. and C. Strachey

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

Milner, R.

[1984] A proposal for standard ML, in: *Proceedings of the 1984 ACM symposium on Lisp and functional programming*, pp. 184–197.

Morris, **F. L.**

[1970] The next 700 formal language descriptions, Unpublished notes, Essex University.

[1973] Advice on structuring compilers and proving them correct in: *Proceedings, ACM symposium on principles of programming languages, Boston*, pp. 144–152.

Morris, J. H.

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

Morris, J. H. and B. Wegbreit

[1976] Subgoal induction, *Comm. ACM*, **20**, pp. 209–222.

Moschovakis Y. N.

[1969] Abstract first order computability I, *Trans. Am. Math. Soc.*, **138**, pp. 427–464.

[1971] Axioms for computation theories – first draft, in: *Logic colloquium 1969*, edited by R. O. Gandy and C. M. E. Yates (North-Holland, Amsterdam) pp. 199–255.

[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.

[1984] Abstract recursion as a foundation for the theory of algorithms, in: *Computation and proof theory. Proceedings, Logic Colloquium, 1983, Part II*, edited by M. Richter, et. al., Lecture notes in mathematics, no. 1104 (Springer, Berlin).

Moses, J.

[1970] The function of FUNCTION in Lisp or why the FUNARG problem should be called the environment problem, Massachusetts Institute of Technology, Project MAC, AI-199.

Mosses, P.

[1982] Abstract semantic algebras! in: *Proceedings, IFIP TC2 working conference on formal description of programming concepts* (North-Holland, Amsterdam).

[1984] A basic abstract semantic algebra, in: *Semantics of data types, international symposium, Sophia-Antipolis, June 1984, proceedings*, edited by G. Kahn, D. B. MacQueen, and G. Plotkin, Lecture notes in computer science, no. 173 (Springer, Berlin) pp. 87–108.

Mosses, P. and D. Watt

[1985] Pascal: A-semantics, Computer Science Department, Aarhus University (draft of work in progress).

Pitman, K.

[1983] *The revised Maclisp manual*, Laboratory for Computing Science, Massachusetts Institute for Technology, MIT/LCS/TR-295.

Platek, R. A.

[1966] *Foundations of recursion theory*, Ph.D. thesis, Stanford University.

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.

[1978] The category of complete partial orders: a tool of making meanings, Lecture notes, summer school on foundations of artificial intelligence and computer science, Pisa, 1978.

[1981] A structural approach to operational semantics, Aarhus University, DAIMI FN-19.

Reynolds, J. C.

[1970] Gedanken - A simple typeless language based on the principle of completeness and the reference concept, *Comm. ACM*, **13**, pp. 308–319.

[1972] Definitional interpreters for higher-order programming languages, in: *Proceedings, ACM national convention*, pp. 717–740.

Sato, M. and T. Sakauri

[1984] Qute: a functional language based on unification, in: *Proceedings of the international conference on fifth generation computer systems '84, Tokyo*, pp. 157–165.

Scherlis, W. L.

[1980] *Expression procedures and program derivation*, Ph.D. thesis, Stanford University.

Scott, D.

[1973] Models of various type free lambda-calculi, in: *Logic, methodology and philosophy of science IV*, edited by P. Suppes, et. al. (North-Holland, Amsterdam) pp.157–187.

[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.

Smith, B. C.

[1982] *Reflection and semantics in a procedural language*, Ph.D. thesis, Massachusetts Institute of Technology.

Steele, G. L. Jr.

[1978] Rabbit: a compiler for Scheme, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Technical Report 474.

[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.

Strachey, C. and C. P. Wadsworth

[1974] Continuations: a mathematical semantics for handling full jumps, Oxford University Computing Laboratory, Technical Monograph PRG-11.

Talcott, C.

[1983] Seus reference manual, unpublished.

Thatcher, J. W., E. G. Wagner, and J. B. Wright

[1980] More advice on structuring compilers and proving them correct, in: *Semantics directed compiler generation, proceedings of a workshop in Aarhus*, edited by N. Jones, Lecture notes in computer science, no. 94 (Springer, Berlin)

Turing, A. M.

[1936] On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, ser. 2, **42**, pp. 230–265.

Wegbreit, B.

[1975] Mechanical program analysis, *Comm. ACM*, **18**, pp. 528–539.

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.

Weyhrauch, R. W.

[1980] Prolegomena to a theory of formal reasoning, *Artificial Intellignce*, **13**, pp. 133–170.