INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106
Marks, Scott Chesser

OPTIMIZATION OF FUNCTIONAL PROGRAMS

Rice University

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Ph.D. 1982
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ☑.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ☑
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _________ lacking when material received, and not available from school or author.
12. Page(s) _________ seem to be missing in numbering only as text follows.
13. Two pages numbered _________. Text follows.
14. Curling and wrinkled pages _____
15. Other __________________________________________

University
Microfilms
International
RICE UNIVERSITY

OPTIMIZATION OF FUNCTIONAL PROGRAMS

by

SCOTT CHESSER MARKS

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

K. W. Kennedy
Professor of Mathematical Sciences
Chairman

R. S. Cartwright
Assistant Professor of Mathematical Sciences

J. R. Junk
Professor of Electrical Engineering

HOUSTON, TEXAS
MAY, 1982
The functional programming style describes computations concisely. The applicative nature of functional programs allows careful analysis of these calculations for optimization. Efficient execution can be achieved by replacing recursive definitions with iterative implementations. The simplified form of functional programs allows effective use of denotational semantics techniques. The same analysis shows a method for implementation as parallel computations using message sending protocols. Networks of distributed processors can be designed in a straightforward way from these results.
The ideas contained herein reflect only dimly the fruit of many seeds planted by those with whom I was privileged to share the community of research. In addition to my committee, others deserving of special recognition include Dr. Scott Warren for a decage of advice, encouragement and friendship, Ms. Velva Power for keeping me organized while doing the same for many others, the National Science Foundation whose Grant NSF MCS77-24093 dealt with the physical realm, and above all my wife Donna, whose support and understanding made it all worthwhile.

Scott Marks
May, 1982
1. Introduction

Optimization efforts assume that programs as written specify more computation than necessary to achieve the desired results. These extra calculations arise from usage of constructs of greater generality than is required for the specific problem as well as from redundant computations introduced by the programmer through error or to embody invariants used to prove the program correct. This work considers optimization of programs written in a functional style. We keep as a goal implementation techniques which exploit the possibilities for parallel computation inherent in such programs.

This chapter discusses the motivation for our initial interest in this work which stems from John Backus' 1977 ACM Turing Award lecture [Back78], the pursuit of parallelism in computation, and a long-standing interest in message-passing systems as a low-level model for computation in general. We also discuss here the overall approach in which we examine various models for functional program computation, discuss implementations, and explore transformations both of high-level programs and of their implementations. Finally, we outline the results of this work which include a technique for analysis of functional programs using continuations, transformations of functional programs based on data structure representations of continuations, techniques for implementations of functional programs on message-passing systems, and the place of optimization approaches including classical techniques as well as those specific to the proposed implementations.

1.1 Motivation

The functional programming style proposed by Backus which forms the backbone of this work addresses the issues of programming language esthetics, the relationship of languages and computational models and the effects that the simple operational model, applicative reduction semantics model and von Neumann computer model maintain on the psychological approach to the process of programming. These ideas tie in with the need for parallelism to
overcome physical limitations on single processors, the fabrication engineering problems of yields versus complexity, and the issue of reliability in elaborate computational schemes. We argue that the parallelism allowed by a functional programming style fits best with a message-sending scheme in which objects and messages are devices supporting modularity, in which Petri nets and their interpretations as data-flow machines have indicates promise as well as a first cut at a semantic definition, and in which the trend toward networks and distributed computation finds a natural home.

From the point of view of a professional career which spanned a period in which hundreds of computer programming languages were designed and having participated in the design of the first widely accepted "high-level" language FORTRAN, Backus speaks with considerable authority to the issue of programming language esthetics. Clarity of exposition of intent is the primary concern for a language which must be used by humans as well as computers. A language which achieves this esthetic goal almost necessarily must be easy to verify as a result of simple proof rules and a propensity for simple mental interpreting using abstraction of sub-part behavior. These qualities dictate a limitation on the number of basic concepts in such a language; this limitation in turn requires that the basic concepts be independent of one another and capable of meaningful combination in the fashion known as "orthogonality".

The attempt to define a language exposes the effect that the computational model addressed by the language has upon its form. Any language must speak of some basic "things" whose interrelationships the language describes. In both overt and covert ways the properties of the computational model that a computer language describes affect the constructs present in a language and the semantic definition of the properties of those constructs.

The simple operational models derived from early experience with machine language coding for the earliest computers were anticipated by Turing and are manifested today in the automata theoretic models of Turing machines and their equivalents. These models have as
their advantage a concise and useful set of basic concepts. Their explicit storage is the basis of history sensitive behavior. The storage is of the most elementary form, as are the internal states, which results in the individual computational actions being small state transitions. This results in obscure programs which are difficult to verify or to use as a basis for deductions of the properties of the behaviors of the programs or of the capabilities of the system as a whole. The profound results in these areas form a testament to the persistence of the researchers who devised them.

In contrast to the simple operational models stand the applicative models of the \( \lambda \)-calculus and combinator theory, the manifestation of these ideas in pure Lisp, and the functional programming systems proposed by Backus. The basic concepts of these systems share the property of conciseness and utility in proofs. Unlike the simple operational models, these have no storage concept, and hence cannot be history sensitive. In the place of states and state transitions the applicative models propose reduction semantics of timeless equivalence. With care in construction programs written in languages founded on the applicative model can be clear.

The von Neumann computer provides the basic concepts for the third and most prevalent computational model. Conventional programming languages reflect this model with distressing clarity. The functional units of these computers typically have a large number of featured behaviors which form a complex interacting basis which make proofs concerning those behaviors difficult. Such computers have storage, usually of several different kinds, and exhibit extremely history sensitive behavior in the name of efficiently. Their fetch-execute cycle is a conceptually simple type of state transition mechanism; however the states involved are so complex that most of the possible benefit of this simplicity is lost. The programming languages and resultant programs for such computers vary in clarity and complexity. A common theme in the language designs is the attempt to fit closely to the particular version of
machine underneath which results in programs designed more for the computer that runs them than the humans which design and read them. Programs for these models which exhibit clarity, like any clear programs, do so because of a locally good fit between the problem and the language; for von Neumann machines such clear programs are generally concerned with low-level concepts such as graphs, and languages which make construction of clear programs easy do so at an execution penalty.

1.1.1 Parallelism

Conventional languages ultimately may only be replaced when the von Neumann processors which they model become obsolete. A machine architecture which exhibits large-scale parallelism will need to avoid the word-size connection between the computational process and its storage, the so-called "von Neumann bottleneck". Parallelism is an attractive alternative to increasing centralized complexity for many reasons.

All computational processes are limited by the information transfer time inside their physical realizations, the "speed of light". As we fabricate larger and larger integrated circuits attempting to minimize this effect, we run into limitations arising from heat dissipation. The number of external connections to each chip grow at least with its word size, and the interconnections within the chip become more complex. Complexity is the enemy of yield, since the density of defects on a given silicon wafer is relatively constant, increasing circuit area increases the chance of including a bad spot.

Parallel computation in circuits of lower individual complexity offers another way out, as well as allowing calculations designed for greater reliability. A reconfigurable network of parallel computations which operate as interchangeable parts forming a complex whole can be designed to survive the death of one of its sub-units. The failure rate of the individual simpler chips will be lower since the product of junction failure rate for that fabrication technology times the number of junctions will be smaller in the simpler circuits.
1.1.2 Message-sending

The third idea motivating this work is that of message-sending as a model for computation. An object-oriented architecture employs objects and messages as modularity devices in a more concrete version of the abstract data type plan. It achieves information hiding by keeping the state of computational objects internal and private to them. Messages to the objects request changes of state as well as revelation of whatever version of state the object wishes to display. Since the semantics of such a computational scheme is defined by the behavior of objects in response to messages [Hew74], different objects which respond to the same message patterns in similar ways can provide alternate implementations with a compatible modular interface.

Another approach to parallelism related to object orientation is the Petri net model and its currently fashionable interpretation, the data-flow machine. Petri net theory provides a skeleton for results concerning liveness and deadlock derived from the topology of network interconnection. These results are limited by the deliberate omission of interpretation. The data-flow application of this theory interprets transitions as computational nodes and place markers as data tokens. The semantic characterization of this model is complicated by the choice of synchronous activity of operators. This choice interacts with the non-determinate nature of the network, resulting in sequences of complex distributed states being the natural semantic domain. Apart from being a difficult framework with which to work, this approach may cause relevance problems when asynchrony due to the speed of information transfer within the network becomes significant.

Nonetheless, some form of network with distributed computation seems unavoidable in the future of computing. As the data-flow approach recognizes, some encoded form of messages are passed within any computational structure. Networks provide the potential for a combined computational and physical solution to problems by using feedback from the physical world with which the computation interacts. Furthermore, no one yet suggests that
computers will ever overtake the ability of men to pose problems too large for them.

1.2 Approach

Any work in optimization follows a well-trodden path, defining computational models and their corresponding implementations, and examining transformations of the implementations to derive heuristics for improvement. We will first examine various models of the computational process for functional program application. We will use a high-level model consisting of the usual reduction semantics for applicative calculations. A low-level model of computation as message-passing will be used when we desire to reveal properties of primitive functions, discuss the calculations implied by functional forms, or in general to expose the inner working of the calculations for further analysis.

The particular implementations we have in mind parallel these computational models. In discussion of source-to-source translation optimizations we rely on an implied high-level implementation on a processor designed to execute functional programs which has the primitive functions together with structural operations as its instructions. Instruction timing in this model remains as a parameter which should not need to be closely examined to determine whether translations are in fact desirable, since such timing assignments would be ad hoc at best. We will follow the algorithms for source-to-source translations with translations to a low-level message-passing implementation. We assume a processor which implements this model in a straightforward way. Here again, timing remains a parameter which should not need close examination.

The real meat of optimization work comes in exploring transformations at various levels to find improved methods of calculation. We will consider control and data flow methods equivalent but preferable to those programmed in the obvious way or resulting from simple-minded implementation. In addition, we will briefly examine the simplification techniques which apply when large projects are constructed from general-purpose subparts. In passing we
note that the vastly simpler nature of functional programs when compared with their conventional counterparts allows much deeper analysis using symbolic evaluation. A form of optimization which is always difficult to apply is the use of alternate implementation algorithms which tend to result in combinational blowup of the translation process, as in compilers for complex machines of the VAX family.

1.3 Results

We summarize here the main results of this work. We adapt a technique for analysis of general recursive programs using continuations to functional programs in particular, including the forms of recursion which are unique to Backus' suggested functional programming system. This analysis leads first to methods for translations of functional programs based on data structure representations for continuations. We also use the same analysis technique to derive implementation of functional programs in message passing systems of multiple parallel computational node. Both of these translations reveal opportunities to apply optimizations both of the classical kind and specific to message passing systems. We first use these optimizations in demonstrating the utility of these translations, and later consider them in a systematic way.
2. Semantics

This chapter outlines the ideas of program semantics we use in defining the meaning of functional programs and in using these definitions in analysis.

When programs are understood only as execution sequences on machines, operational semantics provides a formalization of this low-level understanding. Here we interpret programs as order codes, sequences of instructions for an underlying machine model. The degree of understanding we achieve is that which can be derived from considering the operations of the underlying machine as transformation rules on a large, complex machine state.

Since the operational approach is not designed with mathematical tractability in mind, we contrast it with the techniques of denotational semantics. We interpret programs as mathematical partial functions, with the underlying model a mathematical one of abstract domains chosen to match the concepts of the language involved. The program components again define transformation rules, but instead of executing them to discover their effect, we can describe them mathematically.

2.1 λ-calculus

A concrete semantic domain is Church's λ-calculus. While basically a formulation of recursive function theory, when supplemented with some concrete basic data the λ-calculus transforms into the pure version of the programming language Lisp. Translating to the λ-calculus has an advantage as a semantic technique which derives from its proven completeness and the existence of formal models.

Finally, we discuss the denotational semantic concept of continuation which plays a prominent role in our analysis techniques. Continuations arise in the denotational semantic treatment of languages with jumps, where programs are placed in a semantic domain of continuation transformers. The interpretation of continuations as data structures and as connections to
following calculations forms the basis of our translation algorithms.

2.2 Operational semantics

This section reviews operational semantics. We discuss programs as order code, sequences of instructions for a high-level machine which when executed generate sequences of state modifications. The underlying machine model features a large, complex store altered by simple, low-level primitive operations whose control structure is the classical sequence-with-jumps, rendering a definition of the overall program effect implicit. The transformation rules of an operational semantic definition of programs interprets commands in the programming language as patterns of instructions, with the state of an execution defined by the contents of the store; when registers are included as part of the store, the control flow maintained in a program counter becomes part of that store as well.

2.2.1 Program as order code

Semantic definitions of programming languages construct the meaning of a program from the meanings of the components. To define a program as an order code, the underlying machine model should be one which closely fits the language constructs. The primitive operations of the language correspond to commands executed by the machine. The control flow explicated by the program is implemented by control flow commands. The effect of a program is just a sequence of state modifications. Here the state is the values of variables, plus the value of the program counter, registers, flags, etc. An operational semantic definition simply defines an interpreter which modifies the state once per cycle as program steps are executed. While there may be considerable structure in the program as originally written, by the time it is interpreted within the operational semantic domain, the overall structure is present only implicitly in the execution sequence.
2.2.2 Underlying machine model

The exact nature of an underlying machine model in an operational semantic framework varies according to the language being characterized. Since the languages characterized by this method historically have been conventional von Neumann languages, this model usually has a large, complex store. Heterogeneously within this store one finds values of various types, with differing properties. There is usually a control stack of some sort which, together with the program counter, represents the future calculations still pending; as we will see later, this is just the continuation for the current state. The operational interpreter accesses this store by some type of addressing, although this usually takes the form of "retrieve (or alter) the value of the following variable" rather than a lower level physical address. Since the primitive operations reflect the primitive concepts of the language, in the common case of definition of a conventional language these simple operations effect individual small changes to the store; depending on the level of machine model, these may actually require some form of address calculation, particularly in the case of array operations. Since the control structure is basically that of sequences of operations in basic blocks with connected by jumps, we find it difficult to get an overall picture of the effect of a program. The presence of a control stack with return addresses which may be obeyed or ignored complicates this task even more.

2.2.3 Transformation rules

The transformations rules used in operational semantics derive from the interpretation of commands as instruction patterns affecting the program state which, along with control, is modeled in the store.

The degree of understanding achieved by translating commands in the programming language to patterns of instructions of the underlying model is a subjective matter. Regardless of the level of detail of the machine model, operational semantics tries to expand the constructs of the programming language under consideration to instructions of that model. If the
machine model has the same operations as the language, then no "understanding" has been accomplished by the semantic explication, because the operations executed by the machine are no more comprehensible than the notation used in the language. If the machine model has simpler operations than the language, then we claim to understand the language through the semantic explication because we claim we understand the simpler operations. However, in this case we must comprehend the effect of the more complicated combination of simpler things.

Operational definitions model the state of the computation as the contents of an abstract store. When deciding on a particular scheme for this modeling we must decide which of several different representations we wish to use for the same data. An exhaustive approach will choose several operational models of increasing levels of detail, modeling data at the highest level as elements of an abstract data type, whereas at the lowest level modeling them as bit patterns. Since the store is often relatively unstructured or "flat", all values used in a program scope will in general be present at least throughout that scope. We may include in the model a form of garbage collection after scopes by translating the "leave scope" construct to include garbage collection instructions. Languages which have explicit deallocation of storage will have these constructs translated to similar low-level model instructions. We will find nonetheless that the operational semantic model will be cluttered by the presence of dead values which remain as part of the defined store contents.

Since registers are part of the storage model, the control flow of the operational interpreter in determined by the contents of the store as well. We stress that any form of translation of a language to machine code for any particular machine is a (usually not very useful) operational semantic definition. Even in a higher level model the program counter is present, although sometimes factored out in a separate "component" of the store from the variable value modeling parts. When constructing an operational definition of a language which includes procedure
calls, especially recursive ones, it becomes necessary to define a control stack. Even if this stack, like the program counter, remains in a separate object from the variable value storage area, it is part of the store contents which the interpretation must consider at each step. We do have the freedom to choose representations for the part of control which is bundled in the store which are not quite so closely linked to implementation techniques, but these nonetheless form part of the store which is considered by the interpreter.

2.3 Denotational semantics

Denotational semantics takes the approach that the proper place for analysis is within the domain of established mathematics. We try to define the meaning of a program as a mathematical object which we can subject to rigorously established manipulation techniques. To this end, we try to define a program as a function with an underlying model of a mathematical nature, and with transformation rules defined mathematically as well.

2.3.1 Program as function

We choose the function as the concept to model programs since we expect the output of a program to be completely determined if all the influences on that output are taken into consideration. Composition of other functions whose nature we claim to understand comprises the only method of defining the function which models a given program. Since the semantics is to be derived from the structure of the program we must assign to each construct a function which models it. Essential to this approach is replacing the concepts of successive modifications to a store object with the sequence of distinct states that store would assume, considering each new set of contents to be a different store, where the functions which are composed to form the model for the program under consideration each calculate a new store value from that passed to them as parameter.
2.3.2 Underlying mathematical model

We strive to keep the underlying model within strictly mathematical forms, although this sometimes approaches a kind of "mathematical engineering" rather than an elegant construction. This mathematical model is built upon a given set of undefined basic domains, typically including integers and truth values. Also given as part of the denotational framework is a set of primitive functions defined outside the model and considered as understood prior to modeling any particular program. By choosing simple basic domains and studying the properties of the primitive functions we find that the models we construct for programs have tractable mathematical properties which are useful not only in defining the effect of a program but also in proving facts about the programs behavior.

2.3.3 Transformation rules

The transformation rules for this model follow this general mathematical theme. The commands and other constructs of the programming language are interpreted as patterns, not of instructions of some machine model, but as functions from the given set of primitive functions. As we noted, the state is a value, usually some complex Cartesian product of the basic domains and function spaces defined on those domains. We model control flow as function calls. Some of the most important objections to the denotational semantic framework arise from this fact. First, we think of function evaluation as a timeless reduction kind of step rather than the sort of thing we could make efficiency statements about. Another objection is raised concerning the complex mathematical structures necessary to model programs containing goto statements, error escapes and similar disorderly constructs. This second objection can be partially answered by pointing out the difficulty understanding such constructs presents to any semantic framework, including the intuitive one used by the human programmers. Furthermore, the powerful if complex structure used to model these constructs turns out to be useful as an analytic tool as well; it is the primary tool for most of the results our work.
2.4 \( \lambda \)-calculus as a semantic definition

Church's \( \lambda \)-calculus represents one of the earliest efforts in semantic definition of the meaning of computational activity. Constructed on a small set of basic definitions, it exhibits completeness of definitional power for the set of partial recursive functions, and has several established concrete mathematical models of some power for deriving its properties.

2.4.1 Basic definitions The central concept is that of a \( \lambda \)-expression which defines a function as an expression composing other functions acting on formal parameters. The operation acting on \( \lambda \)-expressions is application, in which a \( \lambda \)-expression followed by a list of actual parameters is replaced by the body of the \( \lambda \)-expression with the actual parameters copied into the positions of the corresponding formal parameters in the body. One can include a set of basic domains as well, although in the fashion of set theorists it is possible to define the elements of these domains as particular functions as well. For instance, truth values can be defined as:

\[
\text{true} = \lambda \, a \, b \, . \, a \quad \text{false} = \lambda \, a \, b \, . \, b
\]

We might observe that the both the actual parameters to these truth expressions will have been evaluated prior to selecting the appropriate one. Packaging the actual parameters in further \( \lambda \)-expressions which will be applied after the selection offers a way out of this. In this framework the truth values will look like:

\[
\text{true}_f = \lambda \, a \, b \, f \, . \, f \, a \quad \text{false}_f = \lambda \, a \, b \, f \, . \, f \, b
\]

2.4.2 Completeness

It is possible to set up within the \( \lambda \)-calculus version of the constuctions used in recursive function theory which model all the partial recursive functions. In this brief survey we can not go into the details of the construction. The approach taken can be described simply as choosing \( \lambda \)-expression models for recursive function concepts. This is essentially the inverse
of a denotational semantic definition of the \( \lambda \)-calculus as a programming language.

2.4.3 \( T\)-OMEGA model

To establish that the \( \lambda \)-calculus is not of vacuous content, we can define concrete mathematical models of \( \lambda \)-expressions and their reductions. The first such models were quite complex exercises in lattice theory and did not provide much insight into the workings of \( \lambda \)-calculus computation. Two models have been constructed based on the natural numbers, which themselves have set-theoretic models such as the Peano construction. The P-OMEGA model interprets elements of the set of subsets of the integers as elements of the \( \lambda \)-calculus. Since this power set has a natural lattice based on set inclusion, it is possible to prove the completeness results by using lattice-theoretic approaches. As the work in denotation semantics became mature researchers realized that complete partial orders, essentially lattices without a "top" element, present a more natural framework for programming language semantic definitions. The \( T\)-OMEGA model [Plot78] is a universal domain in which \( \lambda \)-calculus modeling appears more straightforward. We can think of this domain as either the Cartesian product of denumerably many copies of the truth-value complete partial order composed of \( \top \), true and false, or alternatively as the set of pairs of disjoint subsets of natural numbers.

2.5 Continuations

The orderly world of \( \lambda \)-expression application and reduction avoids the messy concepts of conventional programming languages such as jumps. To deal with these ideas, the concept of continuations is introduced to the denotational semantic framework.

2.5.1 Denotational semantics of jumps

We actually find a sequence of increasingly troublesome constructs which motivate continuations. The simplest of these is the problem of error values which are dealt with in real programming languages by system intervention, such as indexing outside the bounds of an
array, dividing by zero, or dereferencing an invalid pointer value. Local jump (within a scope) is a control flow construct which can be handled by fixpoint theory in a data-flow analysis basic block strategy. The mathematical definition is messy, however. Much more intractable is the problem of arbitrary gotos and recursive evaluation, with their interaction in the form of gotos out of a recursive invocation to an arbitrary outer level of recursion being one of the messiest.

2.5.2 Programs as continuations: transformers

To dig out we consider programs as continuation transformers. We look upon programs as functions from input values to ultimate answers. The latter are uninterpreted but are meant to model output and other effects not necessarily within one of the domains manipulated by the program itself. Included in the set of answers are things like core dumps resulting from improper execution, printed results, and bells ringing on an interactive console. A continuation is simply a function from one of those recursive domains handled by the program to the sets of answers. Programs themselves are modeled as continuations for their inputs. An important dual space construction arises when we observe that if continuations are mappings from object values to answers, then objects can be considered mappings from continuations to answers.

2.5.3 Interpretation of continuations

The importance of continuations lies not only with their elegance in handling troublesome programming language constructs but also with their intuitive meaning. In data-flow analysis we try to elicit the nature of futures for values computed within programs to decide whether a value will subsequently be needed, and if so what alternate representations could be used for that value. The continuation a program defines for a value at a certain point in the program is precisely the mathematical statement of that future. Label values are continuations, since a goto that label represents mapping the state of the program execution at the time of the goto
to the final answers produced by the program subsequent to the jump. The only problem with continuations is that halting problem considerations forbid the existence of a closed-form solution for continuations in the general case. Nonetheless, we can represent a continuation as "the following immediate further actions followed by whatever comes after". By using a data structure to represent that portion of the continuation that is tractable we can bring that continuation into the data domain where optimizing transformations can be applied.
3. Functional programming

This chapter outlines the ideas behind functional programming as presented by Backus in [Back78]. We examine the basic concepts of the objects of the functional programming system and the functions which manipulate them. We discuss the operational semantics of functional programs, including how to compute the value of an application, and present an interpreter which constitutes a complete operational semantic definition. We consider the denotational semantics of functional programs. Since the functional programming systems is mathematically oriented, we handle the denotational semantics of functional programs in a cursory fashion, surveying the domain of values, the use of auxiliary functions, and interpretations of primitive functions and functional forms. We discuss the algebraic laws of functional programs which outline relationships between functional forms involving composition and construction, composition and condition, other forms with composition, condition and construction, etc. Finally we discuss some of the simple expansion theorems which help us understand the behavior of functional programs, including the basic definitions and the linear expansion, recursion and iteration theorems.

3.1 Basic concepts

The most basic concept of the functional programming style is that it is build upon a small number of basic concepts. In this section we first discuss objects, including atoms, sequences and the special object \( \perp \). We also consider the more important basic concept of functions, discussing their general properties, surveying the set of primitive functions, their combination in functional forms, and the programming technique of defined functions.

The choice of a set of basic objects determines the subject of manipulations of a functional programming system. The FP system we will discuss uses a minimal set of atomic objects, sequences of objects, and the special object \( \perp \).
We choose as atoms enough objects to give the functional programming system non-trivial domains without considering extra concepts such as strings which would simply clutter up the discussion. We include the set of integers denoted by numerals among the atoms. Uninterpreted symbols, including what Lisp calls atoms, identifiers and special symbols, are also atoms. We use a particular special symbol ° to denote the empty sequence. Another special symbol holds the place of unimportant values. Among the identifier type symbols, we distinguish the pair T and F for use as Boolean values, returned by predicate functions and expected by conditional forms.

We construct a non-atomic value from already existing objects by enclosing them as elements of a sequence. The pointy brackets < and > surround a list of objects to denote sequences. Since functional programs do not have variables, the contents of the bracketed list will always be objects, so that this constructions is not dynamic. The special symbol ° denotes the empty sequence, as noted above, and is the only sequence which is also an atom. We will also use the term lists for sequences for variety.

Denotational semantics presents the domains of the functions which specify program behavior in the form of complete partial orders with a unique element as the least defined in all orderings. To make the connection with denotational semantics as close as possible, the special object ° is included among the set of functional programming objects. ° is unique in that it is the only object which is neither a sequence nor an atom, and is used to represent undefined values. To keep the cpo of objects flat, improper sequences (with ° as an element) are disallowed, and any notation using the brackets < and > which includes ° in its list of components is interpreted to represent the same object as °. ° is also used as the "value" of divergent computations, which implies that ° is a "non-value" as opposed to a special value such as or °. The usual less-defined-or-equal relation on the resulting cpo is extremely simple, since disallowing improper sequences insures that all pairs of which neither is ° are
Incommensurate; thus all ordered chains of objects are of finite length, in fact no longer than two.

As the name implies, functional programming is primarily concerned with functions. The primitive functions, functional forms and defined functions all share useful general properties.

Each function maps a single object to another single object. We meet the usual need for a multiple-parameter function by applying a function to a sequence whose components correspond to the parameters of conventional versions of the same function. The only operation applicable to any function is application; we write the \( f : x \) to mean the application of the function \( f \) to the object \( x \). Another general property shared by every function is that the result of any application of any function to \( \bot \) is guaranteed to be \( \bot \). All primitive functions have this property, and all functional forms are defined to preserve this property if their component functions have it.

The primitive functions combined with the atomic objects and functional forms give a particular functional programming system its flavor. We necessarily define primitive functions outside of the FP world; for instance, since there is no concept of "set" within our functional programming system, we cannot define the predicate "atom" which tells us whether a non-\( \bot \) object is in the set of predefined atomic objects. We define primitive functions by specifying their result upon application, often using a conditional notation similar to the conditional functional below. We must stress that this is not programming within the FP system, but is simply definition of the system. The selector functions pick out a particular sequence element. In [Back78] Backus denotes a selector with a number parameter in a different font, but herein we denote it with an ordinal name, such as \( \text{first} \). Selectors return \( \bot \) when applied to insufficiently long sequences, or non-sequence objects; they are not dynamic, but it is possible to define an "index" function within the FP system which achieves that effect. The tail function \( \text{tl} \) drops the first element of a sequence, and thus is defined for any non-null lists,
returning \( \bot \) otherwise. The identity function \( id \) takes each object to itself, thereby being \( \bot \)-preserving in a degenerate way. The predicate function \( atom \) tests for membership in the set of pre-defined atomic objects; it must return \( \bot \) when applied to \( \bot \), but returns \( \text{F} \) when applied to any other non-atomic object. The function \( eq \) is defined on pairs, returning \( \text{T} \) if the pair is composed of identical elements. The strictness of sequence construction means that \( eq: \bot \bot = eq: \bot = \bot \). The function \( \text{null} \) returns \( \text{T} \) when applied to \( \phi \), and returns \( \text{F} \) when applied to any other non-\( \bot \) object. \( \text{rev} \) returns a sequence backwards, and is undefined on non-sequences. Two other structural functions, \( \text{distl} \) and \( \text{distr} \) make lists of pairs from singleton-and-list pair arguments, returning \( \bot \) if the argument is not a pair or if the non-singleton (the second element for \( \text{distl} \), the first for \( \text{distr} \)) is not a list. The function \( \text{length} \) is defined only on sequences as the number of elements. The usual arithmetic operations are defined as functions which work only on pairs of numbers; we denote them as \( +, - \), \( \times \) and \( :+ \); they are defined on all pairs of numbers, except that for any object \( x \), \( :+ < x \text{ } 0 > = \bot \). The usual Boolean operations are also available as primitive functions, with \( \text{and} \) and \( \text{or} \) defined on pairs of elements either \( \text{T} \) or \( \text{F} \), and \( \text{not} \) defined on similar singletons. We define the structural function \( \text{trans} \) on rectangular lists-of-sequences, which returns the transposed sequence-of-lists if all the sequences were of the same length, but returns \( \bot \) when applied to any other type of argument. The functions \( \text{apndl} \) and \( \text{apndr} \) are defined under the same conditions as \( \text{distl} \) and \( \text{distr} \), but merely append the singleton to the front or back respectively of the list pair element. Occasionally we have need for the right selectors such as \( \text{rfirst} \), \( \text{rsecond} \), etc., which select from the right end of sequences. We can treat lists as circular with the rotate functions \( \text{rotl} \) and \( \text{rotl} \), which like other list manipulation functions are defined to be \( \bot \) when applied to non-lists. Although the above functions are defined within [Back78], we can add other primitive functions to this list so long as they are suitably well-behaved. An example would be \( \text{shuffle} \) and \( \text{unshuffle} \), which might be defined only on even-length sequences. Alternatively we might define \( \text{pairup} \), \( \text{split} \) and \( \text{flatten} \), and then program \( \text{shuffle} \) and \( \text{unshuffle} \).
As rich as the set of primitive functions is, we will need to do something other than merely apply them to particular objects. Functional forms denote new functions formed from other functions and/or objects as parameters. The most commonly used functional form is composition, written for example as \( f \circ g \), meaning apply \( g \) first, then apply \( f \) to the result. The composition form takes any number of component functions, written \([f_1 \cdots f_n]\), and defines a sequence of results when each component function is defined. Another common functional form is condition, written like \( p \rightarrow f \); \( g \), which denotes \( f : x \) or \( g : x \) when \( p : x \) evaluates to \( T \) or \( F \) respectively. We can use any particular object \( y \) as a parameter to the Constant functional to produce a function \( \overline{y} \) defined to return \( y \) when applied to any non-\( x \) object. We adapt from the APL reduction operators more general forms Insert left and Insert right which when combined with the function parameter \( f \) are written \( \backslash f \) and \( / f \), respectively. The function \( \alpha \) must be defined on pairs, whereupon the Insert form reduces its sequence argument to a single result by working on successive pairs. Insert forms are also defined on single-element sequences as the first selector function; if \( f \) has a unit element of the appropriate side, the corresponding Insert form is defined on \( \phi \). \( \alpha f \) works on sequence arguments by returning the sequence of results of the application of \( f \) to each component of the argument.

Functions defined on pairs can be Curried with an object parameter using the Binarytounary functional, so that \( \mu f \) denotes the same function as \( f \circ [\overline{\mu} \ id] \). The simple semantics of functional programs make the definition of the While functional form a simple fixpoint. Taking two parameter functions \( p \) and \( f \), the form while \( p \ f \) is defined to be the same as \( p - (\text{while} \ p f) \ o f \ ; \ id \).

We use defined functions for notational convenience and recursive self-reference. The notation \( \text{Def} f \equiv \ E \), where \( E \) is a functional form possibly involving \( f \), denotes a strictly textual definition which allows for recursion. The rule for evaluating applications of defined functions is simple textual replacement, applying the right hand side in the place of the
defined function, with no name scoping rules involved due to the absence of variables and function parameters.

3.2 Operational semantics of functional programs

In this section we present an informal operational semantics for functional programs. We must discuss how to compute the value of an application in the cases of primitive functions, functional forms and defined functions. We discuss the role of an interpreter in operational semantics, including handling of some details by the interpretive system and modeling of objects, and present a Rosetta Smalltalk [Warr79] implementation which comprises a more formal operational semantics.

Any operational semantics for functional programs describes how to compute the value of an application as a series of steps leading from the original application expression to a final object result; this sequence need not be of finite length, in which the resulting "value" is assigned \( \bot \), but of course this is uncomputable in general. The steps break down into four cases, of primitive functions, functional forms, defined functions, and \( \bot \) as a value. We compute the value of an application of a primitive function according to its definition outside the FP system; all these primitive function computations terminate. We compute the value of an application of a functional form according to its definition as part of the FP system, with the actual parameters of the form as used replacing the formal parameters in the form definition. We compute the value of an application of a defined function by replacing the defined function by its definition. If the argument of the application is \( \bot \), it may be returned immediately as the value since all functions are \( \bot \)-preserving. This optimization also stops the "take-a-step" loop since \( \bot \) is an object in the FP system. \( \bot \) is returned by applications which are undefined or for which this operational loop does not terminate. Thus it is exactly the "undefined" value, or "non-value"; in a useful system most terminating functions should be total, with appropriate error values filling out the undefined spots, so that for example we might define
Any interpreter forms an operational semantic definition of a language in terms of the machine model used by the language in which the interpreter is written. Its meaning function is simply the composition of the interpretation function for constructs being interpreted with the meaning function of the semantics for the implementation language. Whether this is a useful semantics depends on the "cognitive distance" between the interpreted and implementation languages and on the usefulness of the implementation semantics. Some details of an interpretive implementation are handled by the interpreting system; in the example interpreter below, evaluation of atoms to replace definiens with definition, input of arguments and output of results are handled without further definition. The cognitive distance depends mostly on whether some concepts in the implementation language closely model related concepts. For instance, in this interpreter numbers are interpreted directly, without attempting to model arbitrary size integers. List constructors defined outside the FP interpretation are used to build lists which model those of the FP system directly. We give the Rosetta Smalltalk [Warr79] (abbreviated RS) implementation below. To bridge the cognitive gap we extend RS by defining a Class Function whose objects will implement FP functions. An instance variable of each Function object indicates whether it is a primitive or defined function or functional form. We write RS code for the bodies of primitive functions. We achieve the effect of functional forms by more RS code which applies parameter functions and constructs arguments and result. We get a close semantic fit by implementing FP objects directly as RS objects; that is, there is no Class FPObject which distinguishes them, and thus primitive function bodies can use the full message-sending polymorphism of RS itself. The dynamic parsing used in RS allows a powerful form of syntactic extensibility which resulted in the FP system as implemented executing examples typed in directly from this work and from [Back79] with only minimal lexical transliteration. For instance, we implement only Insert Right (as Insert),
using / for +.

Table 3.1

"Backus Functional Programming implementation",
  "Scott Marks 1980".

@Bottom ← (@Bottom ← Class new title ← "Bottom") new.
Bottom forget @(isnew) answer @(ts a1) by @(eq Bottom a1)

@phi ← @Q!
@T ← yes!
@F ← not

"Class Function",
@Function ← {(Class new title ← "Function"
             edict ← @unpair patimu)
         idict ← @(Form body)
         tdict ← @b f g p x y z)
         cvar unpair ← (Class new new)
         cvar pairnu ← (Class new new)
         answer @(isnew primitive b))
         by @(Form ← @primitive, @body ← b. self)
         answer @(isnew Composition f) (g))
         by @(Form ← @Composition, @body ← pairnu f g. self)
         answer @(isnew Construction (f))
         by @(Form ← @Construction, @body ← b. self)
         answer @(isnew Condition p) (f) (g))
         by @(Form ← @Condition, @body ← pack p + pairnu f g. self)
         answer @(isnew Constant (b))
         by @(Form ← @Constant, @body ← b. self)
         answer @(isnew Insert (b))
         by @(Form ← @Insert, @body ← b. self)
         answer @(isnew Applyall (b))
         by @(Form ← @Applyall, @body ← b. self)
         answer @(isnew Binairequinary (f)) (x))
         by @(Form ← @Binairequinary, @body ← pack f + pack x. self)
         answer @(isnew While p) (f))
         by @(Form ← @While, @body ← pack p + pack f. self)
         answer @(isnew Definition (b))
         by @(Form ← @Definition, @body ← b. self)
         forget @(isnew)
         answer @(print)
         by @(disp ← 60. Form print.
       disp ← 56. body print. disp ← 62. self)
         answer @(ts b) by @(eq Function b)
         answer @(body) by @(body)
         answer @(Form) by @(Form)
Function

\[ \text{answer} @ (f : b) \]

\[ \text{by } @ (x \text{ is Bottom} => \text{Bottom}) \]
\[ \text{Form } = \text{ @primitive } => \text{ (body eval)} \]
\[ \text{Form } = \text{ @Composition } => \text{ (body1 : body2 : x)} \]
\[ \text{Form } = \text{ @Construction } => \]
\[ (\@b \leftarrow \text{ List new @p } \rightarrow \text{ body length}. \]
\[ \text{repeat} \]
\[ (0 = p => \]
\[ (\text{done with b}) \]
\[ (b[p] \leftarrow \text{ body}[p] : x) \text{ is Bottom } => ?r \]
\[ (\text{done with Bottom}) \]
\[ @p \leftarrow p - 1) \]
\[ \text{Form } = \text{ @Condition } => \]
\[ (\text{eq } T @p \leftarrow \text{ body1 : x } => \]
\[ (\text{body2 : x}) \]
\[ \text{eq } F p => \]
\[ (\text{body3 : x}) \]
\[ \text{Bottom}) \]
\[ \text{Form } = \text{ @Constant } => \text{ (body)} \]
\[ \text{Form } = \text{ @Insert } => \]
\[ (x \text{ is List } => \]
\[ (0 = @p \leftarrow (\@b \leftarrow x + @0)) \text{ length } => \]
\[ (\text{Bottom}) \]
\[ @x \leftarrow b[p]. \]
\[ \text{repeat} \]
\[ (1 = p => \]
\[ (\text{done with x}) \]
\[ @x \leftarrow \text{ body : pairu b[@p } \leftarrow p - 1] x) \]
\[ \text{Bottom}) \]
\[ \text{Form } = \text{ @Applytoall } => \]
\[ (x \text{ is List } => \]
\[ (\@b \leftarrow \text{ List new @p } \leftarrow x \text{ length}. \]
\[ \text{repeat} \]
\[ (p = 0 => \]
\[ (\text{done with b}) \]
\[ (b[p] \leftarrow \text{ body : x}[p]) \text{ is Bottom } => \]
\[ (\text{done with Bottom}) \]
\[ @p \leftarrow p - 1) \]
\[ \text{Bottom}) \]
\[ \text{Form } = \text{ @Binarytoinary } => \]
\[ (\text{body1 : pairnu body2 : x}) \]
\[ \text{Form } = \text{ @While } => \]
\[ \text{repeat} \]
\[ (\text{eq } F @p \leftarrow \text{ body1 : x } => \text{ (done with x)}) \]
\[ (\text{eq } T p => (\@x \leftarrow \text{ body2 : x}) \]
\[ (\text{done with Bottom})) \]
\[ \text{Form } = \text{ @Definition } => \]
\[ (\text{body eval : x}) \]
\[ \text{disp } \leftarrow \text{ "Corrupt Function" } \leftarrow 13. \text{ Bottom}}) \]
"primitive functions".

@id ← Function new primitive @x1

@tl ← Function new primitive
   @ (x is List =⇒ (0 < x length =⇒ (x[2 to x length]) Bottom) Bottom)

@isatom ← Function new primitive @ (x is List =⇒ (0 = x length) yes)

@ispair ← Function new primitive @ (unpair)

@iseq ← Function new primitive
   @ (unpair =⇒
      @ (y is Boolean =⇒ (eq y z)
         @ (y is List =⇒
            @ (p = 0 =⇒ (done with yes)
               @ (y[p] = z[p] =⇒ (@ p ← p − 1)
                   (done with no))
            @ y = z)
            Bottom)
   @ y in List =⇒ (0 = y length) F)

@isnull ← Function new primitive @ (x is List =⇒ (0 = x length) F)

@reverse ← Function new primitive
   @ (x is List =⇒
      (b ← List new @ p ← x length.
       x each y do (b[p] ← y. @ p ← p − 1).
      @ b
      Bottom)

@distl ← Function new primitive
   @ (unpair =⇒
      @ (z is List =⇒
         (b ← List new @ p ← z length.
          p = 0 =⇒ (done with b)
          b[p] = pair y z[p].
          @ p ← p − 1))
      Bottom)
      Bottom)
@distr -- Function new primitive
@{unpair =>
  {y is List =>
    @{b => List new @p => y length,
      repeat
        {p = 0 => {done with b},
          b[p] => pair y[p] z,
          @p => p - 1})
      Bottom
    }
  }
Bottom}

@length -- Function new primitive @{x => x length, Bottom}
@+
@- -- Function new primitive @{unpair as Number => (y + z) Bottom}
@* -- Function new primitive @{unpair as Number => (y * z) Bottom}
@
@< -- Function new primitive
@{unpair as Number => (0 < z => (y / z) Bottom) Bottom}

@trans -- Function new primitive
@{x =>
  {0 < x length =>
    {y => x length,
     @b => List new @z => x[1] length,
     repeat
       z = 0 => {done} b[z] => List y, @z => z - 1,
     repeat
       {y = 0 =>
         {done with b},
         x[y] length => @z => b length =>
         {done with Bottom}
     repeat
       {x = 0 => {done} b[z][y] => x[y][z], @z => z - 1},
     @y => y - 1})
    phy
  }
Bottom}

@and -- Function new primitive @{unpair as Boolean => (y and z) Bottom}
@or -- Function new primitive @{unpair as Boolean => (y or z) Bottom}
@not -- Function new primitive @{x => not x} Bottom}
@roll — Function new primitive
@ (x is List =>
  (0 < x length => (x[2 to x length] + x[1 to 1]) phi)
  Bottom)

@retr — Function new primitive
@ (x is List =>
  (0 < x length =>
   (x[x length to x length] + x[1 to x length - 1]) phi)
  Bottom)

@apndl — Function new primitive
@ (unpair => (x is List => (pack y) + z) Bottom) Bottom)

@apnds — Function new primitive
@ (unpair => (y is List => (y + pack z)) Bottom) Bottom)

include "selector"

@first — selector 11
@second — selector 21
@third — selector 31
@last — rselector 11
forget (selector rselector)
"Functional Forms".

Function
   answer @(! (g)) by @((Function new Composition self g)
   answer @(! => ! (l) ; (g)) by @((Function new Condition self f g))

@! <-
   (Class new odict <= @(ConstructBuilder)
      ever ConstructBuilder <-
      (Class new tidct <= @(!) odict <= @(!list)
         answer @(!isnew) by @(@(list <= List now O. self)
         answer @(!) by @((Function new Construction list)
         answer @(!a1)) by @(@(list <= list + pack a1.
            self)) ) new.

I is?
   forget @(!isnew)
   answer @(!) by @((ConstructBuilder new)
   answer @(!print)
      by @(!disp <= 60. self is? print. disp <= 62.)

@Constant <- Class new tidct <= @(!y) now.
Constant is?
   forget @(!isnew)
   answer @(!y) by @((Function new Constant y))

@Insert <- Class new tidct <= @(!f) now.
Insert is? forget @(!isnew) answer @(!f)) by @((Function new Insert f))

@Applytall <- Class new tidct <= @(!f) now.
Applytall is?
   forget @(!isnew)
   answer @(!f)) by @((Function new Applytall f))

@Binyteunary <- Class new tidct <= @(!f x) now.
Binyteunary is?
   forget @(!isnew)
   answer @(!f) (x)) by @((Function new Binyteunary f x))

@While <- Class new tidct <= @(!p f) now.
While is?
   forget @(!isnew)
   answer @(!p) (f)) by @((Function new While p f))

@Def <- Class new tidct <= @(DEFF b) now.
Def is?
   forget @(!isnew)
   answer @(!((DEFF = (@b))) by @((DEFF = Function new Definition b))
"unpair and pairnu are utility objects for handling pairs".

Function cvar unpair is? tdict ← @c)
                           forget @t@snew)
                           answer @t()         by @t(x is List =)
                              (x length = 2 => (y ← x[1], @t z ← x[2], yes) no)
                              no)
                           answer @t(as (c)) by @t(salt =) {y is c and z is c) no)

Function cvar pairnu is? tdict ← @t(y z)
                           forget @t(isnew) answer @t((y) (z)) by @t((pack y) + (pack z))}
3.3 Denotational semantics of functional programs

Denotational semantics places the meaning of the constructs used in a language in the world of mathematical functions which describe the results computed by programs in the language. This style of semantics is extremely simple for our functional programming system since its orientation is \textit{a priori} mathematical. A description of the approach to construction of such a definition is given in this section. The domain of values is simply the set of FP objects. It is a recursive domain since components of sequences may be any object other than $\bot$, but it is simple since its ordering is totally flat. The standard auxiliary semantic functions such as \texttt{Cond} and \texttt{BuildSeq} are assumed. The interpretations for primitive functions are simply the mathematical functions described in the system definition for them. The interpretations of functional forms are equally straightforward. For instance, the meaning of \texttt{while}\,$g$ is $\texttt{fixpoint}(f)$ where $f = \texttt{Cond}(p, (f \circ g), \texttt{id})$. 
8.4 Algebraic laws of functional programs

In this section we summarize some of the algebraic laws which functional forms obey and which allow us to analyze and to transform functional programs.

<table>
<thead>
<tr>
<th>Algebraic Laws of Functional Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.1 ((f_1 \cdots f_n) \circ g = (f_1 \circ g \cdots f_n \circ g))</td>
</tr>
<tr>
<td>I.2 ((\alpha f) \circ [g_1 \cdots g_n] = [f \circ g_1 \cdots f \circ g_n])</td>
</tr>
<tr>
<td>I.3 (f \circ [g_1 \cdots g_n] = f \circ [g_1 / f \circ [g_2 \cdots g_n]]) when (n &gt; 1)</td>
</tr>
<tr>
<td>I.4 (f \circ [f x] = (\text{but} f x) \circ g)</td>
</tr>
<tr>
<td>I.5 (\text{first} \circ [f_1 \cdots f_n] \leq f_1)</td>
</tr>
<tr>
<td>I.5.1 ([f_1 \circ \text{first} \cdots f_n \circ \text{nth}] \circ [g_1 \cdots g_n] = [f_1 \circ g_1 \cdots f_n \circ g_n])</td>
</tr>
<tr>
<td>I.6 (f \circ [f_1 \cdots f_n] \leq [f_2 \cdots f_n] ) when (n &gt; 1)</td>
</tr>
<tr>
<td>I.7 (\text{dist} \circ [f \circ [g_1 \cdots g_n]] = [[f_1 \circ g_1] \cdots [f \circ g_n]])</td>
</tr>
<tr>
<td>I.8 (\text{apndl} \circ [f \circ [g_1 \cdots g_n]] = [f \circ g_1 \cdots g_n])</td>
</tr>
<tr>
<td>I.9 ([\cdots \text{I} \cdots ] = [\text{I}])</td>
</tr>
<tr>
<td>I.10 (\text{apndl} \circ [f \circ g \circ (\alpha f) \circ h] = (\alpha f) \circ \text{apndl} \circ [g h])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Composition and condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>II.1 ((p - f; g) \circ h = p \circ h - f \circ h; g \circ h)</td>
</tr>
<tr>
<td>II.2 (h \circ (p - f; g) = p - h \circ f; h \circ g)</td>
</tr>
<tr>
<td>II.3 (\text{and} \circ [p \circ g] - f; \text{and} \circ [p \text{ not} \circ q] - g; h \leq p - (q - f; g); h)</td>
</tr>
<tr>
<td>II.3.1 (p - (p - f; g); h - p - f; h)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Composition and others</th>
</tr>
</thead>
<tbody>
<tr>
<td>III.1 ((\bar{x}) \circ f \leq \bar{x})</td>
</tr>
<tr>
<td>III.2 (f \circ \text{id} = \text{id} \circ f = f)</td>
</tr>
<tr>
<td>III.3 (\text{first} \circ \text{distr} \leq [\text{first} \circ \text{first} \circ \text{second}])</td>
</tr>
<tr>
<td>III.4 (\alpha (f \circ g) = (\alpha f) \circ (\alpha g))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition and construction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV.1 ([f \cdots (p - g; h) f_n] = p - [f_1 \cdots g \cdots f_n]; [f_1 \cdots h \cdots f_n])</td>
</tr>
</tbody>
</table>
3.5 Expansion theorems

In this section we review some expansion theorems which help us explain the workings of functional programs and to prove transformations correct. These theorems are proved in [Back79, pp. 627-628].

Many recursively defined functions may be understood using the clarifying expression derived from the following.

Theorem: Let $f$ be a solution of $f = p \circ g$; $Q(f)$ where $Q(k) = h \circ (i \circ j)$, and $p$, $g$, $h$, $i$ and $j$ are given functions. Then

$$f = p \circ g; p \circ j = Q(g); \cdots p \circ j^n = Q^n(g); \cdots$$

and we also have

$$Q^n(g) = h \circ (i \circ j \cdots i \circ j^{n-1} \circ g \circ j^n).$$

As a simple application of this we can show the correctness of a recursive factorial function. We define the function thus:

```plaintext
Def rfact ≡ eq 0 - \bar{1}; × o [id rfact \circ sub 1]
Def eq 0 ≡ eq o [id 0bar]
Def sub 1 ≡ - o [id 1bar]
```

Then $rfact$ satisfies the hypothesis of the recursion theorem with $p = eq 0$, $g = \bar{1}$, $h = \times$, $i = id$ and $j = sub 1$, so we have:

$$rfact = eq 0 - \bar{1}; \cdots$$

$$eq 0 \circ sub 1^n - Q^n(\bar{1}),$$

where

$$Q^n(\bar{1}) = × o [id$$
Functional programming

\[ id \circ sub1 \cdots \]

\[ id \circ sub1^n-1 \]

\[ \bar{1} \circ sub1^n \]

Now for any number \( n \), if \( (eq0 \circ sub1^n) : x = T \) then \( x-n \) and \( Q^n(\bar{1}) : n \to <n \cdot n-1 \cdots 11> \), so

\( tfact : x = x=0-1; \cdots x=1-n \cdot (n-1) \cdots \times 1 \times 1 \)

which is precisely the factorial function.

A corollary of the recursion theorem gives a simple expansion for many iterative programs:

**Theorem:** Let \( f \) be a solution of \( f = p - g \circ f \circ k \). Then we have

\( f = p - g \circ k - h \circ g \circ k; \cdots p \circ k^n - h^n \circ g \circ k^n; \cdots \)

Functions of this form are called iterative since they break down into essentially two While loops. The first evaluates \( p : x, p : k ; x, \cdots \), until \( p : k^n : x = T \) where all the previous values were \( F \), counting the number \( n \) of iterations. The second loop calculates \( h^n : g : x \) as the value of the application. As an example of the use of this theorem, let us define an iterative factorial function and prove it correct. We use an accumulator to collect the multiplied part of the factorial value as a future for the remaining calculations. Since the future of the decremented first argument will be to be multiplied by the accumulator, the identity future value for the initial accumulator contents is \( 1 \).

\[ \text{Def } ifact = ifactc \circ [id \ 1 \bar{1} \bar{bar}] \]

\[ \text{Def } ifactc = eq0 \circ \text{first} - \text{second} \; ; \; ifactc \circ [sub1 \circ \text{first} \times] \]

This satisfies the conditions of the iteration theorem with \( p = eq0 \circ \text{first} \), \( g = \text{second} \), \( h = id \) and \( k = [sub1 \circ \text{first} \times] \). Since \( h = id \), we can see that this function would need only one While
loop for completely explicit iterative form. Thus we have

\[ ifacte = p - g; \cdots p \circ k^n - g \circ k^n \]

since \( h = id \). Since we know that the argument of \( ifacte \) is a pair, we need only show that for all \( n > 0 \) \( k^n = [a_n, b_n] \) where the functions \( a_n \) and \( b_n \) are

\[ a_n = sub1^n \circ first \]

\[ b_n = \times \circ [sub1^{n-1} \circ first \cdots sub1 \circ first \ first second] \]

which we would expect from the form of the recursion theorem above. This holds immediately for \( n = 1 \) from the definition of \( k \) since

\[ \times \circ [first \ second] = \times \circ [first \ second] = \times. \]

For \( n > 1 \), we have

\[ k^{n+1} = k \circ k^n = [sub1 \circ first \times] \circ [a_n, b_n] = \]

\[ [sub1 \circ sub1^n \circ first \times \circ [sub1^n \circ first \cdots sub1 \circ first \ first second [[] = \]

\[ [sub1 \circ sub1^n \circ first \times \circ [sub1^n \circ first \cdots sub1 \circ first \ first second [[]] = \]

\[ [a_{n+1}, b_{n+1}]. \]

Now for any number \( n \geq 0 \), all of \( eq0:n, eq0:sub1:n, \ldots, eq0:sub1^n:n \) will be \( F \) and \( eq0:sub1^n:n \) will be \( T \), so \( ifact:n \) will be \( second:k^n:n = b_n:n = \times<12 \cdots n-1 \ n \ 1> = n! \).
4. Analysis of functional programs using continuations

This chapter deals with the analysis of functional programs into constituent actions so as to follow closely the disposition of calculated results. This analysis focuses primarily on recursion in functional programs. We particularly emphasize analysis which leads to means of removal of recursion through transformations [Ausi73, Bird77, Burs77, Darl74, Leve77, Stee78a, Stee78b], particularly those which introduce explicit stacking and restoration of program state. In Chapter 6 we will consider an architecture similar to data-flow machines in which recursion presents problems, requiring (at least a simulation of) the growth of new pieces of machine to handle recursive invocations. Implementors of functional programming systems on conventional architectures should find these transformations interesting as well, since they discuss how to handle forms of recursion peculiar to functional programs.

4.1 Factoring programs into compositions

A skilled functional programmer employs the same structured programming techniques as one in more a conventional framework. In particular, he decomposes a large problem into smaller subproblems whose solutions he merges into an approach to the original task. The conventional programmer uses decomposition into subroutines and sequential actions as the most common forms of subdivision. The functional programmer employs his analogous techniques with defined auxiliary functions modelling subroutines and functional composition modelling sequential execution. Here we consider programs which arise from the latter technique. In particular, we wish to pursue as far as possible the approach of denoting the continuations for values computed by programs using functions within the functional programming style. Later we will use an extended notation involving the $\lambda$-calculus to handle more complex cases and to separate the continuations more precisely.
4.1.1 Selectors and state value sequences

Conventional languages often employ sequential action to accomplish a group of small changes to the state of the program as a whole. These Algol-like languages use many "word-size" variables whose values taken collectively with the current program counter define the state of the computation. An "elegant" conventional program will have a minimum number of such variables whose values are primarily independent from one another. The program counter appears implicitly in a functional program as well. The rest of the program state must be explicitly collected into a single value, since a functional program and all its parts work on single arguments producing single results. This single value must be sufficiently complex to contain the information conventionally spread among the large number of small variables, and thus is often a large sequence with components corresponding to variables or groups of variables of the Algol-like program. Functional programs which have been decomposed into compositions of simpler functions model the small changes to a large, diffuse state which would be accomplished by a sequence of commands. Such a simpler function often will dissect the sequence, picking out the components of interest, and return as its value a sequence quite similar to its argument, with only a few components different. Sometimes the result sequence will contain fewer components, achieving the effect of combining the values of multiple variables into one without the attendant (admittedly often implicit) allocation and deallocation of space.

This use of sequences to contain state information implies that the mathematical notion of conventional variables as naming functions from a large state to particular values becomes more explicitly realized as selector functions are used to pick out components of the state vector. The variable names in an Algol-like language perform a notational service in analysis of the calculations by assisting in the decomposition of the analysis in a manner analogous to the decomposition of the original problem. Even in mechanical analysis the programmer's vari-
ables become the starting point for further consideration. The "native names" of the primitive selector functions such as *first* are not at all mnemonic to a human analyst. The unavoidable multiple uses of the same selector obfuscate intent, particularly when a changing state vector places the same pieces of information in different sequence slots for successive steps of the computation.

In languages with variables, the useful lifetime of values is often less than that dictated by formal definitions. For instance, Algol68 local values enjoy a scope which extends to the end of the block in which the identifier denoting the value is declared. This may include several statements after the last use, a region where the value is dead. Since functional programs do not have variables, but instead maintain equivalent state information in the single parameter being acted upon, such values simply may be dropped from the parameter sequence when no longer needed. Several steps in a functional composition may use the same data, if only to pass it unaltered to the successive steps as a component of the state-modelling sequence. The new computational results present imply a new view of the state of the computation and may require a new notation to describe that state. This new notation would be implemented in a conventional language by new values for variables. In the case of small changes, an existing variable would be updated; if the value was of a new mode or the old value of the existing variables were still useful, a new variable would be initialized. In functional programming the new value occupies a component slot in the state vector, and a new name function would be used to access it. Other new name functions might be necessary as well merely to access pre-existing values carried along in a new position. The functional programmer finds a Hobson's choice between using the non-mnemonic positional names (*first, second*, etc.), keeping track outside the program of which data are where, or inventing a multiplicity of special-purpose names for the selector functions. Despite the advantages of explicitly acknowledging the functional nature of names in this functional programming style, the conventional language con-
cept of an updated variable provides a connection between the successive notational conventions of successive computational steps which assists in understanding what being computed. FAD [Mart80] offers a solution of syntactic sugar in the form of local function names with specific ranges of application which ease this problem somewhat. This allows a selector function used as a name to be redefined at the boundary of the definition of a "larger" function, so that the same name can be used to refer to the datum as it appears in different component positions in the state-modelling sequence. We will adopt a variant of this notation as can be seen in the examples to follow.

4.1.2 Separation of actions

Analysis of a program composed of many computations must start by separating out the individual computational activities. In a functional program, the computations are of two kinds. Bookkeeping activities which do not compute essentially new results are easy to handle. We will call combinations of selectors and the identity function with construction functionals structural functions. An example function is \[\text{[second first [second third]]}\], where the state-modelling sequence is rearranged, but no new information is present. The non-structural functions actually compute something. The square root function and the eq0 function are examples of non-structural functions.

We will attempt to elicit the "future" of each value a program computes so as to decide whether it need be computed and how the computations might be optimized. A problem arises when computations are proceeding in parallel. Consider the function \(\text{RMS}\) defined as

\[
\text{Def RMS} \equiv \text{sqrt} \circ \circ (\text{bn} \times 2) \circ [\text{square} \circ \text{first square} \circ \text{second}].
\]

In the application \(\text{RMS}:< A B >\) corresponding to the expression \((A^2 + B^2)/2\), the future of the datum \(A\), roughly "to be squared and added to the value \(B^2\), and then ...", contains within it a mention of the previously computed value \(B^2\), whose computation could perhaps
best proceed in parallel. The symmetric continuation for the value \( B \) paradoxically contains the previously computed value \( A^2 \). Thus our desire to consider each value separately forces a choice of a sequential order for execution. Our choice will be left-to-right unless noted otherwise, although in general there will be several equally good choices available. Whichever order is chosen, we will decompose the given function into compositions of purely structural functions with functions computing a single new value. The structural functions may be bundled in as necessary, but the "computational" functions will be strictly segregated, one computation per function. In the above example, the function \( RMS \) would be broken into a composition of five functions:

\[
\text{Def } RMS \equiv \sqrt{\circ (\mu x + 2)} \\
[\text{square } \circ \text{ first second}] \circ [\text{first square } \circ \text{ second}].
\]

We have broken the parallel computation of the squares of the argument component into two sequential actions. This form expresses the continuations involved in the sense that the calculation of continuations in a simple composition of functions is trivial in the absence of jumps or side effects.

We now examine the famous recursive example program \( \text{tips} \) which counts the leaves in the frontier of a tree. This particular version assumes that each node is either a leaf or a possibly empty sequence of subtrees.

\[
\text{Def } \text{tips} \equiv \text{leaf} \cdot \overline{1}; (\lambda +) \circ (\alpha \text{ tips}) \circ \text{sons}
\]

We observe that this has the correct effect on an empty tree, represented by a root node with no subtrees, since the function \( + \) has the right unit \( 0 \), so \((\lambda +) \circ x = 0\). The function already divides into a sequence of compositions along each conditional branch. We assume that \( \text{leaf} \) and \( \text{sons} \) are defined in an obvious way elsewhere and are primitive for our purposes. The remaining problem is to define the future for the sequence of subtrees created by \( \text{sons}:x \) when
leaf \( x \rightarrow F \). This future breaks in to a number, _a priori_ unknown, of distinct futures for the elements of sons \( x \). Each element will have tips applied to it; the order of application is unspecified, arbitrary, and there is no reason the sub-computations could not be performed in parallel. The resulting subtree tip counts will be assembled into a sequence, say \( y \); the futures are now distinguished, since each count goes in a different sequence slot. This distinction is somewhat lost by the application \( \backslash +; y \) since the associative nature of + allows us any of the possible schemes for adding up the total tip count. The previous remarks hold in a degenerate manner for the case sons \( x = \phi \).

As noted with RMS, to be specific about the individual futures requires that we assume some ordering, simulating serial execution. We will evaluate the subtree counts in left-to-right order, and thus the future of each subtree will have at least two components: those subtrees remaining to be evaluated later must be evaluated, and those counts already obtained must be saved, until all subtrees have been counted, when the total can be calculated.

\[ \text{Def} \ tips \equiv \text{leaf} - \overline{1}; \]

\[ (\backslash +) \circ \text{second} \circ \]

\[ (\text{while} \ not \circ \text{null} \circ \text{first} \ [l \circ \text{first} \ apndr \circ [\text{second} \ tips \circ \text{hd} \circ \text{first}]) \circ \]

\[ [\text{sons} \ \phi]. \]

Taking advantage of the fact that + has the left unit 0 allows us to pass the + into and through the loop:

\[ \text{Def} \ tips \equiv \text{leaf} - \overline{1}; \]

\[ \text{second} \circ \]

\[ (\text{while} \ not \circ \text{null} \circ \text{first} \ [l \circ \text{first} + \circ [\text{second} \ tips \circ \text{hd} \circ \text{first}]) \circ \]
Analysis of functional programs using continuations

\([\text{sons } \emptyset]\).

The sticking point here is the recursive application of tips to the head element of the list of remaining subtrees. The techniques we derive in this and the next two chapters will allow us to reduce this to

\[
\text{def \ } \text{tips } \equiv \text{ second } \circ
\]

\[
(\text{while not } \circ \text{null } \circ \text{first}
\]

\[
(\text{leaf } \circ \text{hd } \circ \text{first} - \text{tl } \circ \text{first } \text{inc } \circ \text{second});
\]

\[
(\text{append } \circ \text{[sons } \circ \text{hd } \circ \text{first } \text{tl } \circ \text{first } \text{second}] ) \circ
\]

\(\text{[[id } \emptyset]\).

An alternative approach pursued in Chapter VI will give continuations a more concrete reality, allowing some continuations to be formed from a conjunction of other named continuations. This analysis technique cannot be employed within the functional programming framework since the names involved are of the parameter variety, not naming functions.

4.1.3 Recursion-loop rolling

We will need to be able to compute continuations for data within functions with embedded recursive calls. We can consider all such function invocations as black boxes, indeed so consider all applications, but in functional program analysis this does not allow us to get very far. We assume that we completely understand the nature of the primitive functions in the sense that we consider their appearance in "closed forms" for continuations acceptable. Many simple functions consist only of combinations of primitive functions where the continuation for any particular value can be written as a simple expression. The presence of recursive calls interferes with the simple analysis of a composition of functions which otherwise would lead to closed forms for continuations. Where a function is only tail-recursive, we consider it to be a
loop and allow the fixed-point operator as part of a continuation expression.

4.2 Analysis using continuations

We will now use an extended notation employing a hybrid of functional programming style with the \( \lambda \)-calculus. That is, we will allow a \( \lambda \)-expression as a continuation denotation, but will encourage it to contain as many functional programming notations within the body as possible so as to retain the feeling of the functional programming style. This is in the spirit of the denotational semantics analyses of [Scot72, Scot77, Stoy77], wherein mathematical objects of greater complexity are introduced to handle concepts such as jumps and side effects, even when simpler techniques would suffice, because the complex formulation parallels more closely the form of the computation specified by the original program.

4.2.1 Closures

In our analysis so far the greatest deviation from the spirit of the function under consideration arises from the need to carry along values which have already been computed pending the computation of other values yet to be calculated. The \( \lambda \)-calculus has the power to denote objects as well as functions, and we will exploit that power now.

4.2.1.1 Definition of closures

Following [Wegb74, Stee76a, Stee76b], we define the concept of the closure of a \( \lambda \)-expression to assist us in the description of continuations. A closure defines a new function \( g \) from an old function \( f \) and a number of free variable-value pairs \( e_1 = e_1, e_2 = e_2, \ldots \) by means of the substitution rules. The functional programming system includes the binary-to-unary functional as a method of explicitly specifying closures. As an illustration, suppose the addition function is \( f = \lambda x y . x + y \). Then the closure of \( f \) with respect to parameter value \( x = 2 \) is the function whose body is the closure of the body of \( f \) with respect to the free variable value \( x = 2 \), equal to the function \( \lambda y . 2 + y \). (In functional programming notation this simple
closure is just $bu\ f\ 2$.)

Closures define a method of partial evaluation of a function with respect to a subset to its parameters. The ability to create closures dynamically and treat the resulting objects as $\lambda$-calculus objects (functions) provides a method for implementing lazy evaluation [Hend77, Vuil74], as noted in [Stee76a, Stee76b, Stee78a, Stee78b]. A closure as a data structure contains the values of the free variables bound in closing. Let us define the $\lambda$-expression

\[
apply = \lambda f x . f : x
\]

A continuation-based calling convention assumes the datum $y$ to be modeled by $\lambda f . f : y$, the closure of the apply function with respect to datum parameter $x$. When closed in the symmetric way, $\textit{apply}$ becomes effectively just the function $\textit{f}$ in its usual form: $\lambda x . f : x$.

4.2.1.2 Environment binding

The concept of closure presupposes related concepts of times and environments. Since the result of evaluating a closure with values for its remaining free variables is defined to be identical to evaluating the original expression with values for all free variables, there must be some further distinction to justify the two ideas. We use the terms binding time and run time to capture this distinction. The intent is that prior to execution of a part of a program, some values such as constants and previously computed results are already available, whereas other values have yet to be examined. The \textit{a priori} values can be combined with the intended execution to define a closure, a special function depending only on the remaining data since all other information has been incorporated in it. Binding time is the point at which this combination occurs, while run time is the point of final application to unspected values. Execution in the normal sense can be viewed as a procession of the binding time-run time dichotomy through the body of the program as some values are derived before the remaining values are acted upon.

Environment in denotational semantics captures the connection between identifiers and the entities which they identify. In languages with variables environment refers to the
connection between the names the programmer uses and the storage locations to which they refer, while state denotes the current value of those variables.

Within the pure \( \lambda \)-calculus the notion of environment is reflected in the choice of value which an expression denotes, particularly when the expression involves free variables bound to one of a set of possible values due to nested invocation of \( \lambda \)-expressions employing different uses of the same identifiers. When closures are included, it is intended that the value associated with a free variable be bound to it permanently, even when the closure is used in an environment in which the free variable would normally refer to a different value. This requires an implementation to preserve the binding environment if closures of the same original expression are to share the generated instructions for that expression. It is usually considered too expensive to generate a modified instruction sequence containing constant references where the original accessed variables, but this may not be the case when a closure is to be used many more times than the function from which it was derived.

The environment of functional programs is deliberately restricted. All functions are statically defined, the only identifiers are those denoting functions and the fixed set of functional forms, and any particular evaluation is concerned with a single unnamed argument. In analyzing the execution sequence created by the application of a functional program, we can separate the individual computations of the program into functions which are combinations of purely structural functions with a single non-trivial computation each. The structural function carries along previously derived results with the current computation in the state-modelling sequence. When we try to separate out the non-trivial part of the computation for further manipulation, the problem of keeping track of these previously derived results arises. This we solve by forming a \( \lambda \)-continuation closure with these results embedded in the body of the \( \lambda \)-expression. These results cannot be merely denoted by functions as they were in the original program. The functions which would denote these results must be applied to the original
argument from which they were calculated, and not the argument to which the continuation is applied, namely the result of the singled-out computation. A first cut might suggest a convention of special functions which would be applied prior to the distinguished computation; this convention falls down when such manipulations as simple substitution of a defined name by its definiens are attempted. What is missing is a scheme for keeping track of the environment in which the closure is to be formed. Fortunately the original functional programs have such a simple state, the single argument, that all that is necessary is to name the argument with which the closure is formed. Further, uniform replacement rules simplify this even further, so that usually only two identifiers are required: one, conventionally \( x \), denotes the original argument with which the continuation is formed; the other, conventionally \( y \), denotes the result computed by the singled-out function.

4.2.1.3 Lexical versus dynamic scoping

These complex issues of name-value association help generate the concept of a variable free functional programming style. Yet if we are to use the \( \lambda \)-calculus as a medium in which to embed our analysis of functional programs, we must press on a bit farther. The two dominant plans for connecting names and values lie at the ends of a spectrum which the concept of closures helps to fill in.

At one end we find dynamic languages which can be thought of as allowing implementations in which all names and values are kept on a list of association pairs which is extended whenever a new scope is entered and trimmed back when such a scope is left. Pure Lisp and the Smalltalk variants Smalltalk-72 [Golb76, Shoc79] and Rosetta Smalltalk [Warr79] use such a scheme. Under dynamic scoping the value used for a free variable in an expression depends on the dynamic execution context of that use; if the expression is the body of a \( \lambda \)-expression, two invocations of the function containing it can cause different values to be associated with the free variable in the expression evaluation.
At the other end of the scoping spectrum we find languages such as the Algol family and Lisp variants like Scheme. In these languages the association between variables and values is made by static examination of the text in which the variable occurs. This means that in the situation above, the free variable in each instance would refer to the value to which it would be associated at the point at which its $\lambda$-expression was defined, not where it was used. These languages allow a simpler, stack-oriented environment management implementation. They also enjoy the advantage of referential transparency, a term used to mean that an expression denotes the same value wherever it occurs.

In any scoping regime where closures are in use care must be taken to insure in some way that the values bound in when a closure is created are still available when it is used. In a dynamic language this may mean copying the relevant parts of the name-value association list; in a static language it may mean implementing some form of otherwise unnecessary indirection in variable evaluation.

4.2.2 Continuation builders

We look at the continuation for a value as being composed of a composition of those functions to be applied to the value during the remainder of the execution of the program with some external system continuation.

Halting problem considerations forbid the existence of a closed form solution for continuations; even functional programs are not immune, since Turing machines are easily simulated. However, some of the initial segment of the future computational sequence for each datum is visible in the local text under consideration. We will analyze what we can, peering into the near future only. Consider an application expression with the external continuation specified such as $g;f:x[R]$. Suppose that we are attempting to analyze the function $f$, and that such an application would arise during the elaboration of an application of $f$, so that this would represent a recursive call. The future for the recursive result $f:x$ is "to have $g$ applied, and
the result passed to the continuation \( R' \). We can express this in \( \lambda \) notation as:

\[
g : f : x \{ R \} = f : x \{ \lambda y . g : y \{ R \} \}.
\]

Here we express that although we may not know what actions \( R \) represents, we do know the immediate future of the value \( f : x \). That future has a layer of \( \lambda \)-expression surrounding the original continuation. Following [Wand80], we use the term \textit{continuation builder} to denote the function which has been composed ahead of the original continuation. Our task remains to find a representation for such knowable parts of continuations.

The lack of computed functions within the functional programming style considerably simplifies the analysis of a typical functional program. This implies that a continuation builder is either a simple (structural plus primitive) or a defined function. This continuation builder composed with the external continuation is the continuation for the result. The external continuation in turn is either the system continuation, or a composition of similar continuation builders with the system continuation. A table containing information about the combining laws of the simple functions can be used with information derived from a particular set of defined functions to simplify the form of continuations, which just means optimize further computations with the argument. The structure of the argument itself often is analyzable from the static program text. For example, selectors limit consideration to sequences with sufficient components, since all further computations will preserve the \( \bot \) derived from application of an invalid selector.

4.3 Categorization of continuation builders

If we limit our analysis to the near future as suggested above, we observe that the continuation for the argument of a small function often consists of a simple composition of some analyzable function with the continuation used as the future of the calculated result. This observation suggests that classification of continuation builders will help us find exploitable
patterns. For instance, a property of continuation builders known to be useful is associativity. If a continuation builder is associative, and has a (one-sided) unit, then an accumulator can be introduced to encode the form of the continuation and its result during the computation.

4.3.1 Trivial continuation builders

The first kind of continuation builder we call trivial. This term includes a simple return, an invocation of a simple function, a simple tail recursion equivalent to a while loop, and certain other tail-recursive situations. In particular, a repeated composition of the called function at its own exit can be handled using a representation which merely keeps track of pending uncompleted autorecursions (loop iterations).

Example - McCarthy's 91-function

McCarthy's famous recursion example is defined in functional programming language as:

\[ \text{Def } f \equiv \text{gt}100 - \text{sub}10; f \circ f \circ \text{add}11 \]

\[ \text{Def } \text{gt}100 \equiv > \circ [\text{id} \text{ 100}] \]

\[ \text{Def } \text{sub}10 \equiv - \circ [\text{id} \text{ 10}] \]

\[ \text{Def } \text{add}11 \equiv + \circ [\text{id} \text{ 11}] \]

We consider an application:

\[ f : x = \text{gt}100 : x - \text{sub}10 : x ; f : f : \text{add}11 : x \]

Next, we include the continuation \( \gamma \) which will use the result:

\[ f : x \{ \gamma \} = \text{gt}100 : x - \text{sub}10 : x \{ \gamma \} ; f : f : \text{add}11 : x \{ \gamma \} \]

We separate the two recursive calls, placing one into the continuation:

\[ f : x \{ \gamma \} = \text{gt}100 : x - \text{sub}10 : x \{ \gamma \} ; f : \text{add}11 : x \{ \lambda y . f : y \{ \gamma \} \} \]
4.3.2 Simple continuation builders

A simple continuation builder is one in which the continuation representation consists only of argument values and the structure which holds them. This suffices when there is a single occurrence of recursive call so that it is only necessary to preserve active data values across the recursive computation.

Example - list reversal

Here we consider a definition for the function reverse, as if it were not supplied as a primitive function. The definition we use is:

\[ \text{Def } \text{rev} \equiv \text{null} \cdot \overline{\Phi}; \text{ apndr } \circ [\text{rev } \circ t \cdot \text{hd}] \]

The embedded recursive call creates the problem here. We must preserve the value \( \text{hd}:x \) until we have calculated \( \text{rev}:tl:x \) to construct the argument \( \langle \text{rev}:tl:x \text{hd}:x \rangle \) to \( \text{apndr} \).

We can look at this as a special case of the following definition:

\[ \text{Def } \text{r} \equiv p \cdot f; g \circ [r \circ s \cdot r] \]

To analyze further, we must consider a particular application:

\[ r:x = p:x \cdot f:x; g:<r:s:x t:x> \]

We embed this in the continuation which will use it:

\[ r:x \{ \gamma \} = p:x \cdot f:x \{ \gamma \}; g:<r:s:x t:x> \{ \gamma \} \]

Next, we expand the false arm of the conditional to expose the recursive call:

\[ r:x \{ \gamma \} = p:x \cdot f:x \{ \gamma \}; r:s:x \{ \lambda y.g:<y t:x> \{ \gamma \} \} \]

To proceed further we translate this back to the particular problem at hand:

\[ \text{rev}:x \{ \gamma \} = \text{null}:x \cdot \overline{\Phi}:x \{ \gamma \}; \text{rev}:tl:x \{ \lambda y.g:<y \text{hd}:x> \{ \gamma \} \} \]
The continuation for the value computed by \textit{rev} will always have the form
\[
\{ \lambda y. \text{apndr}: \langle y \, x_1 \rangle \} \{ \lambda y. \text{g}: \langle y \, x_2 \rangle \} \{ \cdots \{ \gamma \} \cdots \}
\]
with zero or more \textit{\lambda}s. The function \textit{apndr} is constant in these forms, so that the only information needed for an encoding of the continuation is \textit{\langle x_1 \, x_2 \, \cdots \rangle}. We will develop such encodings in the next chapter.

\textbf{4.3.3 Multiple continuation builders}

In the general case there may be multiple continuation builders used within the body of a recursive function. When a function invokes itself or other mutually-recursive functions in more than one place, it is necessary to represent within the continuation data structure which builder is to be employed for given results.

Example - tree walk

An archetypal for multiple self-recursion is a binary tree-walk. We abstract this to the pattern exemplified by the following definition:

\[
\text{Def } f \equiv \rho - a; b \circ [f \circ l \circ f \circ r]
\]

Here we name two functions \textit{l} and \textit{r} to suggest left and right sub-trees, where \textit{p} is a predicate distinguishing leaves from inner nodes, \textit{a} is the action appropriate for leaves, and \textit{b} the computation merging the results of examining the subtrees. Consider the usual application-with-continuation:

\[
f : x \{ \gamma \} := p : x - a : x \{ \gamma \}; b : \langle f : l : x \, f : r : x \rangle \{ \gamma \}
\]

As usual, we expose an innermost recursive call:

\[
f : x \{ \gamma \} - p : x - a : x \{ \gamma \}; f : l : x \{ \lambda y. b : \langle y \, f : r : x \rangle \{ \gamma \} \}
\]

However, in the continuation for \textit{f:l:x} we find another embedded recursive call. We can
expose it:

\[ f: x \{ y \} = p: x - a: x \{ y \}; f: l: x \{ \lambda y . f: r: x \{ \lambda z . b : < y z > \{ y \} \} \} \]

This gets a bit confusing, so to simplify things, we name the two continuations \( \rho_1 \) and \( \rho_2 \), and re-write as:

\[ f: x \{ y \} = p: x - a: x \{ y \}; f: l: x \{ \rho_1 \} \]

\[ \rho_1 = \lambda y . f: r: x \{ \rho_2 \} \]

\[ \rho_2 = \lambda z . b : < y z > \{ y \} \]

Two things need noticing here. First, the continuation \( \rho_1 \) contains a closure utilizing \( x \), the original argument. Second, the continuation \( \rho_2 \) also contains a closure using \( y \), the argument to its context. The next chapter will represent each such continuation by a tag denoting its functionality paired with its argument.

4.4 Functional form iterative continuation algorithm

We now abstract from the above examples a general method for removal of recursion from functional programs. The main motivation for this approach is the work of the next chapter where we endeavor to move recursive calls to a tail-recursive position, whereupon the function may be converted to a while loop.

We imagine that we are working upon the abstract tree of a application of the program with a given continuation. We start by finding an innermost recursive call. The subtree rooted at that call becomes the body of the revised function, with the rest of the original function as the body of a \( \lambda \)-expression which uses the result calculated by that subtree bound to the formal parameter together with the original continuation to become a new continuation for the subtree result. The body of the \( \lambda \)-expression is just the original function with the pruned subtree replaced by the formal parameter. We repeat this process on the derived \( \lambda \)-expression.
until each recursive call is exposed in a tail-recursive position. The nature of the original functional program insures that the each derived continuation is a closure whose bound values depend only on the single argument used by the subtree from which it was derived. Hence we can translate this version using continuations which are \( \lambda \)-expression closures to an isomorphic version in which each recursive call stacks a pair consisting of its own argument plus the distinguished atom which represents its continuation. The other portions of the transformed version will consist of non-recursive calls which apply their given continuation to their results in a similar manner. These calls are translated to calls to a sending function to interpret the effect of the built-up continuation data structure on their results. The sending functions interprets its stack argument as representing the identity continuation if the stack is empty, and otherwise as denoting the continuation whose tag is stacked along with the original argument to be used in calculating the closure to be applied to the newly obtained result.

In essence, there are two phases to the above process. In the first, an application of the original function with a continuation is translated to a simpler application with a more complicated continuation. In the second phase, discussed in the next chapter, the continuations are replaced with a data structure which can be interpreted to implement them. These computations work on the abstract tree of the application, where each node is an FP function, either primitive, simple, or built from those by functional forms. Since primitive and simple functions present no problem, we must specify how to handle a functional form which involves a recursive call.

We take a slight liberty in the translation for the Constant functional form as given below. This form is defined to give a \( \perp \)-preserving function; yet the translation merely passes the constant on to the continuation without checking to see if the argument is \( \perp \). We can fix this up easily if strictness is desired, or simply consider this to be an allowable optimization extension, as noted earlier. In a similar vein, we give only the translation of \( /f \) for functions \( f \)
which have a right unit. Such forms built from functions which do not have a right unit are undefined on \( \phi \). A revision of the translation in this case is straightforward since \( \text{first} \) is conveniently undefined on \( \phi \) as well.

<table>
<thead>
<tr>
<th>Functional form</th>
<th>Continuation version</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Composition</strong></td>
<td>( f \circ g : x { R } ) is already in iterative form</td>
</tr>
<tr>
<td>( g \circ f : x { R } = f : x { \lambda y . \langle y , g : x \rangle { R } } )</td>
<td></td>
</tr>
<tr>
<td><strong>Construction</strong></td>
<td>( \langle f , g \rangle : x { R } = f : x { \lambda y . \langle y , g : x \rangle { R } } )</td>
</tr>
<tr>
<td><strong>Condition</strong></td>
<td>( (p - f ; g) : x { R } = p : x - f : x { R } ; g : x { R } )</td>
</tr>
<tr>
<td>( (f - g ; h) : x { R } = f : x { \lambda y . y - g : x { R } ; h : x { R } } )</td>
<td></td>
</tr>
<tr>
<td><strong>Constant</strong></td>
<td>( \langle y \rangle : x { R } = y { R } )</td>
</tr>
<tr>
<td><strong>Binarytounary</strong></td>
<td>( \langle h , f , z \rangle : x { R } = f : \langle z , x \rangle { R } )</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>( \langle f , x { R } = \langle x , e \rangle { S } ) (( e ) is the right unit for ( f ))</td>
</tr>
<tr>
<td>( S = \lambda y . \langle \text{null} , \text{first} : y - \text{second} : y { R } ; f : \langle \text{rh} , \text{first} : y ; \text{second} : y \rangle { \lambda z . \langle \text{rl} : \text{first} : y , z \rangle { S } } )</td>
<td></td>
</tr>
<tr>
<td><strong>Applyloall</strong></td>
<td>( \langle \alpha , f \rangle : x { R } = \langle x , \phi \rangle { S } )</td>
</tr>
<tr>
<td>( S = \lambda y . \langle \text{null} , \text{first} : y - \text{second} : y { R } ; f : \langle \text{hd} , \text{first} : y \rangle { \lambda z . \langle \text{rl} : \text{first} : y , \text{apndr} , \langle \text{second} : y , z \rangle { S } } )</td>
<td></td>
</tr>
<tr>
<td><strong>While</strong></td>
<td>( (\text{while} ; p , f) : x { R } = x { S } )</td>
</tr>
<tr>
<td>( S = \lambda y . p : y - f : y { S } ; x { R } )</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.** Continuation versions of Functional Forms

In the above examples we handled recursive calls embedded in the functional forms of composition, construction and condition. The methods used are reiterated in Figure 1. In that table \( f \) denotes the function whose recursive calls are to be removed, \( R \) the continuation supplied to the particular recursive call, and \( p , g \), etc. indicate other functions considered simple for this step in the analysis.
The self-referential nature of the Insert, Applytoall and While functional forms results in the necessity to name their continuations so as to let those continuations refer to themselves. We consider this to be justified as an abbreviation for the use of the λ-calculus Y-operator, which provides a means of self-reference by functions defined in terms of the fixed-point of a functional expression involving the function itself in recursive invocation.
5. Representation of continuations in functional programs

This chapter deals with the representation of the results of analysis of functional programs into constituent actions. We particularly emphasize means of removal of recursion through transformations [Ausl78, Bird77, Burs77, Darl74, Love77, Stel78a, Stel78b], particularly those which introduce explicit stacking and restoration of program state. In the next chapter we will consider an architecture similar to data-flow machines [Acko79, Gost79, Kosl78, McGr79, Wats79] in which recursion presents problems, requiring (at least a simulation of) the growth of new pieces of machine to handle recursive invocations. Implementors of functional programming systems on conventional architectures should find these transformations interesting as well, since they discuss how to handle forms of recursion peculiar to functional programs.

In the previous chapter we analyzed a number of functional programs to obtain a form for the continuation builder each specified. Although we can use some of this information directly, it is not expressed in a way which we know how to manipulate. We seek some representation for the continuation which we can access within the optimization framework. This requires us to implement the continuation as a data structure composed of elements of the functional programming system. In this section we re-examine the examples of the previous chapter with the goal of motivating a general algorithm for representing the continuations of functional programs, as well as indicating how the product of that algorithm can be further optimized.

5.1 Continuation interpreters

We can define auxiliary functions which interpret the representation of a continuation to achieve the desired continuation effect. This is simply the observation, most often made in a Lisp environment, that program and data are interchangeable. The continuation interpreter utilizes the data structure used to represent the desired continuation to emulate the functional effect represented. For a given static set of functions, we can easily find a representation for
continuations by using a Pascal-like enumerated type (set of otherwise meaningless identifying atomic symbols) to represent the functions, one per function. We will represent a built-up continuation by a list whose components are either elements of that type, data values which have been computed previously for use in the continuation, or other built-up representations. Functional programming systems do not include functions computed dynamically using the values of functional arguments, although of course these can be programmed by user-defined interpreters. A system which allows such functions can use a representation like the one above and permit user manipulation of the representation as implementation of dynamic functions. This is just the S-expression concept of Lisp with much detail omitted.

The term sending functions will denote auxiliary functions which interpret the represented continuation. We use this name to suggest a communication path for the computed value. Recall the basic motivation for a continuation, in which we consider the argument being sent to the function which then sends its result to the continuation. We realize that when we represent the continuation by a data structure, the operation of "receiving a result" must be implemented by the interpreter for the data structure. This is merely an instance of a typical modular programming scheme in which operations are defined along with the data structures they use to achieve desired computational effects. One can think of the sending function as a special kind of interpreter for continuation data structures. In the world of denotational semantics we find any program reduced in meaning to the continuation for its input values. We treat an interpreter as if it has two inputs: a pre-input, the program to be interpreted, which is stored in a control memory, and a conventional input to the interpretive execution. The situation after the program store is loaded seems to be the same as that of a sending function invoked with an argument representing the continuation. Thus, we conjecture that any interpreter is a sending function, and sending functions are thus not special types of interpreters after all. The sending functions dispose of the values obtained during
recursive invocation. In this way we unify the type of continuation used by the outermost level of invocation (the system continuation, a "real function") with that used by inner calls by choosing an identity function representation which indicates that the outermost call can be terminated and the system continuation used. In the place of the outermost call we employ the continuation version of the function with the identity representation in the continuation parameter slot.

5.2 Characterization of continuation builders

In the next section we give an algorithm for representing the continuation specified by any functional form in a way which removes recursive invocations in any position. To motivate this algorithm in this section we first work several examples which highlight a characterization scheme for continuation builders based on the position and number of recursive invocations.

5.2.1 Trivial continuation builders

The first kind of continuation builder we call trivial. This term includes a simple return, an invocation of a simple function, a simple tail recursion equivalent to a while loop, and certain other tail-recursive situations. In particular, a repeated composition of the called function at its own exit can be handled using a representation which merely keeps track of pending uncompleted autorecursions (loop iterations).

Example - McCarthy's 91-function

McCarthy's famous recursion example was defined in functional programming language as:

\[
\text{Def } f \equiv \text{gt100} - \text{sub10}; f \circ f \circ \text{add11}
\]

\[
\text{Def } \text{gt100} \equiv > \circ [\text{id} \ 100]
\]
Def $\text{sub10} \equiv - \circ [\text{id} \ 10]$

Def $\text{add11} \equiv + \circ [\text{id} \ 11]$

We separated the two recursive calls, placing one into the continuation:

$$f:x \{y\} = \text{gt100}:x - \text{sub10}:x \{y\};$$

$$f: \text{add11}:x \{\lambda y . f:y \{y\}\}$$

We translate this back into the realm of functional programs by using the atom $p$ to represent the continuation. Mimicking a standard implementation technique, we stack the representative of the continuation together with the information needed to resolve any closures, ignoring for the moment that there are none:

Def $f[c] \equiv \text{gt100} \circ x - \text{fsend} \circ [\text{sub10} \circ x \ c];$

$$f[c] \circ [\text{add11} \circ x \ \text{push} \circ [[p \ x] \ c]]$$

Def $\text{fsend} \circ [v \ c] \equiv \text{null} \circ c - v;$

$$\text{finterp}\circ[v \ \text{tag} \circ c \ \text{val} \circ c \ \text{pop} \circ c]$$

Def $\text{finterp} \circ [v \ r \ x \ c] \equiv \text{eq} \circ [r \ p] - f[c] \circ [v \ c];$ ⊤

Def $f2 \equiv f[c] \circ [\text{id} \ \text{newstack}]$

These and future definitions use the following auxiliary functions which define a common implementation for stacks:

Def $\text{push} \equiv \text{apndl}$

Def $\text{top} \equiv \text{hd}$

Def $\text{pop} \equiv \text{tl}$

Def $\text{newstack} \equiv \overline{\phi}$
Since we will in general be stacking two items in a pair, we also define:

\[ \text{Def } \text{tag} \equiv \text{first } \circ \text{top} \]

\[ \text{Def } \text{val} \equiv \text{second } \circ \text{top} \]

Now we consider optimization. There is only one non-trivial continuation builder; hence the continuation interpreter function need not test the stacked atom which indicates which continuation to emulate, and that atom need not be stacked. Further, as noted above, the continuation contains no closed data values, and thus the original argument need not be stacked either. With these changes the function as written would stack a series of phis (empty sequences), wherein the only information would be the length of the stack. Thus we can choose a new representation for the continuation consisting merely of the length of the original stack.

\[ \text{Def } fc3 \circ [x \ c] \equiv \text{gt}100 \circ x - fsend3 \circ [\text{sub}10 \circ x \ c]; \]

\[ fc3 \circ [\text{add}11 \circ x \ \text{inc } \circ c] \]

\[ \text{Def } fsend3 \circ [y \ c] \equiv \text{eq}0 \circ c - y; \ finterp3 \circ [y \ \text{dec } \circ c] \]

\[ \text{Def } finterp3 \circ [y \ c] \equiv fc3 \circ [y \ c] \]

\[ \text{Def } \text{inc} \equiv + \circ [\text{id } \bar{1}] \]

\[ \text{Def } \text{dec} \equiv - \circ [\text{id } \bar{1}] \]

\[ \text{Def } eq0 \equiv \text{eq } \circ [\text{id } \bar{0}] \]

\[ \text{Def } f3 \equiv fc3 \circ [\text{id } \bar{0}] \]

Now we substitute these simple functions with some rearrangement to obtain:

\[ \text{Def } fc4 \circ [x \ c] \equiv \text{not } \circ \text{gt}100 \circ x - fc4 \circ [\text{add}11 \circ x \ \text{inc } \circ c]; \]
\[
\text{not} \circ \text{eq} 0 \circ c - f_c 4 \circ [\text{sub} 10 \circ x \circ \text{dec} \circ c]; \\
\text{sub} 10 \circ x
\]

**Def** \( f_4 \equiv f_c 4 \circ [\text{id} \circ 0] \)

We can change the tail-recursive function \( f_c 4 \) to nested \textbf{while} loops and re-introduce the selectors to eliminate naming functions to obtain a final version:

**Def** \( f_5 \equiv \text{sub} 10 \circ \text{first} \circ \\
(\text{while} \not= 0 \circ c \\
\quad (\text{while} \not= 0 \circ \text{gt} 100 \circ x \circ [\text{add} 11 \circ x \circ \text{inc} \circ c]) \circ \\
\quad [\text{sub} 10 \circ x \circ \text{dec} \circ c]) \circ \\
(\text{while} \not= 0 \circ \text{gt} 100 \circ x \circ [\text{add} 11 \circ x \circ \text{inc} \circ c]) \circ \\
\quad [\text{id} \circ 0]
\)

### 5.2.2 Simple continuation builders

A simple continuation builder is one in which the continuation representation consists only of argument values and the structure which holds them. This suffices when there is a single occurrence of recursive call so that it is only necessary to preserve active data values across the recursive computation.

**Example - list reversal**

Here we consider a definition for the function \( \text{rev} \), as if it were not supplied as a primitive function. The definition we use is:

**Def** \( \text{rev} \equiv \text{null} - \tilde{0}; \text{apndr} \circ [\text{rev} \circ \text{tl} \circ \text{hd}] \)

The embedded recursive call creates the problem here. We must preserve the value \( \text{hd} : \text{x} \)
until we have calculated \( \text{rev} \cdot \text{tl} \cdot x \) to construct the argument \( <\text{rev} \cdot \text{tl} \cdot x, \text{hd} \cdot x> \) to \( \text{apnd} \).

We can look at this as a special case of the following definition:

\[
\text{Def } r \equiv p - f; g \circ [r \circ s \cdot t]
\]

The continuation for the value computed by \( r \) will always have the form

\[
\{ \lambda y . g : <y \cdot t : x_r> \{ \lambda y . g : <y \cdot t : x_s> \{ \cdots \{ y \} \cdots \} \}
\]

with zero or more \( \lambda \)s. As we saw in the last chapter, when the functions \( g \) and \( t \) are constant in these forms,

the only information needed is \( <x_1, x_2 \cdots> \). The well-known solution to representing continuations of this form is to stack the values \( x_i \), with the empty stack representing the continuation \( \{ y \} \) and the stack \( \text{push} : <z \cdot c> \) representing the continuation \( \{ \lambda y . g : <y \cdot t : z> \{ \Delta \} \} \) where \( \Delta \) is the continuation represented by \( c \). Since we will be using an FP object to represent the continuation, we need an interpreter to simulate the continuation application. As in the first example, we have only one continuation builder, and thus need not stack a representative of which continuation builder is being used. Furthermore, the interpretive function is trivial, and can be included in the sending function. We also observe that the continuation closure uses the original argument only to calculate \( t \cdot x \), and thus we can stack only that value instead of stacking the original argument. Having chosen our representation, we can revert to FP definitions:

\[
\text{Def } r2 \equiv rc \circ [id \cdot \text{newstack}]
\]

\[
\text{Def } rc \circ [x \cdot c] \equiv p \circ x - \text{rcsend} \circ [f \circ x \cdot c];
\]

\[
rc \circ [s \circ x \cdot \text{push} \circ [t \circ x \cdot c]]
\]

\[
\text{Def } \text{rcsend} \circ [v \cdot c] \equiv \text{null} \circ c - v; \text{rcsend} \circ [g \circ [v \circ \text{top} \circ c] \circ \text{pop} \circ c]
\]

The definitions above use auxiliary functions for a common stack implementation defined
above. Here we push the values \( t \cdot x \) until \( p \cdot x = T \), whereupon we interpret the built-up continuation. We have converted the function to iterative form, with the continuation interpreter in iterative form also. Thus we can write:

\[
\text{Def } rc \circ [x \ c] \equiv \ rsend \circ [f \circ x \ c] \circ \\
\text{while } not \circ p \circ x \\
[s \circ x \ push \circ [t \circ x \ c]]
\]

\[
\text{Def } rsend \circ [v \ c] \equiv v \circ \\
\text{while } not \circ \text{null} \circ c \\
[g \circ [v \ \text{top} \circ c] \ \text{pop} \circ c]
\]

A moment's reflection leads us to notice that \( rsend \) has a simpler definition:

\[
\text{Def } rsend \equiv \ \lambda g \circ \text{apndl}
\]

using the left-associative form of the insert functional. This in turn can be substituted in the definition of \( rc \) to give us:

\[
\text{Def } rc \circ [x \ c] \equiv \ \lambda g \circ \text{apndl} \circ [f \circ x \ c] \circ \\
\text{while } not \circ p \circ x \ [s \circ x \ push \circ [t \circ x \ c]]
\]

Whereas this form is quite nicely iterative, we can do even better if we know that \( g \) is associative so that \( \lambda g = g \). To proceed, we need the following result:

Suppose we have the following:

\[
\text{Def } j \circ [x \ c] \equiv p \circ x - j \circ [f \circ x \ push \circ [h \circ x \ c]]; c
\]

\[
\text{Def } k \circ [v \ c] \equiv p \circ x - k \circ [f \circ x \ g \circ [h \circ x \ c]]; c.
\]

Then
Representation of continuations in functional programs

\[ I_g \circ j \circ [x \ c] = k \circ [x \ I_g \circ c]. \]  
(1)

Proof (using subgoal induction [Morr77]):

\[ I_g \circ j \circ [x \ c] = I_g \circ \]

\[ (p \circ x - j \circ [f \circ x \ push \circ [h \circ x \ c]]; c) \circ \]

\[ [x \ c] \]  
(defn of j)

\[ = p \circ x - I_g \circ j \circ [f \circ x \ push \circ [h \circ x \ c]]; \]

\[ I_g \circ c \]  
(Law II.2)

\[ = p \circ x - k \circ [f \circ x \ I_g \circ push \circ [h \circ x \ c]]; \]

\[ I_g \circ c \]  
(ind, hyp.)

\[ = p \circ x - k \circ [f \circ x \ g \circ [h \circ x \ I_g \circ c]]; \]

\[ I_g \circ c \]  
(defn of Insert g)

\[ = k \circ [x \ I_g \circ c]. \]

Having finished with that digression, we can continue with the analysis of \( rc \).

\[
\text{Def } rc \circ [x \ c] \equiv I_g \circ apndl \circ [f \circ x \ c] \circ
\]

\[
\text{while not } p \circ x \ [s \circ x \ push \circ [t \circ x \ c]]
\]

\[
= I_g \circ apndl \circ [f \circ x \ c] \circ
\]

\[
\text{while not } p \circ x \ [s \circ x \ push \circ [t \circ x \ c]] \quad (g \ is \ assoc)
\]
Representation of continuations in functional programs

\[-g \circ [f \circ x \cdot /g \circ c] \circ\]

while \(not \circ p \circ x \cdot [s \circ x \cdot \text{push} \circ [t \circ x \cdot c]]\) \((\text{defn of } /g)\)

\[-g \circ [f \circ x \circ (\text{while } not \circ p \circ x \cdot [s \circ x \cdot \text{push} \circ [t \circ x \cdot c])\circ c \circ (\text{while } not \circ p \circ x \cdot [s \circ x \cdot g \circ [t \circ x \cdot c]]) \circ\]

\([x \cdot /g \circ c]\) \((\text{see (1)})\)

\[-g \circ [f \circ x \circ \text{while } not \circ p \circ x \cdot [s \circ x]\]

\(c \circ \text{while } not \circ p \circ x \cdot [s \circ x \cdot g \circ [t \circ x \cdot c]] \circ\]

\([x \cdot /g \circ c]\)

\([-g \circ [f \circ x \circ \text{while } not \circ p \circ x \cdot [s \circ x]\]

\(c \circ \text{while } not \circ p \circ x \cdot [s \circ x \cdot g \circ [t \circ x \cdot c]] \circ\]

\([x \cdot /g \circ c]\)

\[-g \circ [f \circ x \circ c] \circ\]

\((\text{while } not \circ p \circ x \cdot [s \circ x \cdot g \circ [t \circ x \cdot c]]) \circ\]

\([x \cdot /g \circ c]\)

In the above, at (2) we have used the independence of the first part from the value of \(\text{push} \circ [t \circ x \cdot c]\) to eliminate the latter from that state-modelling sequence. At (3) we have used this independence again to move the argument \([x \cdot /g \circ c]\) outside of the construction to be shared by both components of the argument to the outer \(g\). At (4) we used this property one more time to merge the two loops.

Now suppose \(g\) has a right identity \(e\), so that \(/g; \phi = \backslash g; \phi = e\) and \(g; <x \cdot e> = x\) or \(\bot\)

for all \(x\). Recalling from above that we defined \(\tau 2\) to use \(ec\), we have:
The last form of definition for \( rc \) is the result of a well-known transformation which replaces an associative continuation builder with modification of an accumulator initialized to the identity for the building function. What is new here is that this transformation is not simply "pulled from the hat" and then proved correct, but is instead a simple consequence of a result derived independently of the existence of the identity element.

Finally, we return to the list reversal function, our motivation for this extended analysis. Although \( apndr \) is not an associative function, we can use the following simple identity to convert to one:

\[
\text{apndr} \circ [y \ z] = \text{append} \circ [y \ [z]]
\]

Thus in \( rc \) above, we let \( f = \text{newstack} \), \( p = \text{null} \), \( s = \text{tl} \), \( g = \text{append} \), \( t = \text{hd} \) and \( c = \text{phi} \) to obtain:

\[
\text{Def \ rev2} \equiv \text{revc} \circ [\text{id} \ \text{newstack}]
\]

\[
\text{Def \ revc} \circ [x \ c] \equiv \text{null} \circ x - \text{append} \circ [\text{newstack} \ c];
\]
\[
\text{re} \circ [\text{tl} \circ x \; \text{append} \circ [c \; [\text{hd} \circ x]]]
\]

\[
= \text{null} \circ x - c; \; \text{re} \circ [\text{tl} \circ x \; \text{apndr} \circ [c \; [\text{hd} \circ x]]]
\]

We can now substitute to obtain a completely explicit iterative form:

\[
\text{Def rev}2 \equiv c \circ (\text{while} \; \text{not} \circ \text{null} \circ x \; [\text{tl} \circ x \; \text{apndr} \circ [c \; [\text{hd} \circ x]]] \; \circ [\text{id} \circ \xi]).
\]

5.2.3 Multiple continuation builders

In the general case there may be multiple continuation builders used within the body of a recursive function. When a function invokes itself or other mutually recursive functions in more than one place, it is necessary to represent within the continuation data structure which builder is to be employed for given results.

Example - tree walk

An archetype for multiple self-recursion is a binary tree-walk. We abstract this to the pattern exemplified by the following definition:

\[
\text{Def } f \equiv p - a; b \circ [f \circ l \; f \circ r]
\]

Here we name two functions \(l\) and \(r\) to suggest left and right sub-trees, where \(p\) is a predicated distinguishing leaves from inner nodes, \(a\) is the action appropriate for leaves, and \(b\) the computation merging the results of examining the subtrees. As we saw in the last chapter, an explicit form for the continuation-with application is:

\[
f{:x \{y\}} = p{:x - a{:x \{y\}; f{:l}{x \{p_1\}}}
\]

\[
p_1 = \lambda y . f{:r}{x \{p_2\}}
\]

\[
p_2 = \lambda z . b{:<y z> \{y\}}
\]

Several things need noticing here. First, the continuation \(p_1\) contains a closure utilizing \(x\), the
original argument. Second, the continuation $\rho_2$ also contains a closure using $y$, the argument to its context. Third, there are two distinct continuation builders present, so the interpretive function must distinguish between them; stacking the atoms $\rho_1$ and $\rho_2$ to represent them is an obvious choice. Thus we can move back into the functional programming domain with the following definitions:

\[
\text{Def } fc \circ [x \ c] \equiv p \circ x - fsend \circ [a \circ x \ c];
\]

\[
fc \circ [x \ c] \equiv \text{null} \circ c - v;
\]

\[
finterp o [v \ \text{tag} \ c \ \text{val} \ c \ \text{pop} \ a \ c]
\]

\[
\text{Def } finterp \circ [v \ t \ x \ c] \equiv eq \circ [t \ \overline{p_1}] - fc \circ [r \circ x \ \text{push} \circ [\overline{p_2} \ v] \ c];
\]

\[
eq c \circ [t \ \overline{p_2}] - fsend \circ [b \circ [x \ v] \ c];
\]

\[
I
\]

\[
\text{Def } f2 \equiv fc \circ [x \ \text{newstack}]
\]

An important point here is that when the interpretive function makes a recursive call, the "original argument" it stacks for closures in the continuation for the result of that call is the result $v$ it received, and not the previous "original argument" $x$ which it was allowed to use. Hence when the continuation $\rho_2$ is being simulated, the $\lambda$-parameters $y$ and $z$ correspond to the $finterp$ arguments $x$ and $v$. This is actually easier to keep straight mechanically than it is to read, since in the $\lambda$-closure form we use new parameter names to keep things straight, whereas $finterp$ always uses the same name (x) for the old value popped off the stack for constructing closures, and the same name (v) for the newly obtained recursive call result.

As usual, we consider optimization. Although there is more than one continuation builder,
there are only two, and the set \{T, F\} can be used to some advantage to represent the two continuations instead of \{p_1, p_2\} to yield a third version:

\[
\text{Def } fc3 \circ [x \ c] \equiv p \circ x \ - \ fsend \circ [a \circ x \ cl];
\]

\[
f_{c3} \circ [l \circ x \ push \circ [[T \ x] \ c]]
\]

\[
\text{Def } fsend \circ [v \ c] \equiv \text{null} \circ c - v; \ f_{\text{interp}} \circ [v \ tag \circ c \ val \circ c \ pop \circ c]
\]

\[
\text{Def } f_{\text{interp}} \circ [v \ i \ x \ c] \equiv t - fc3 \circ [r \circ x \ push \circ [[F \ v] \ c];
\]

\[
f_{send} \circ [b \circ [x \ v] \ c]
\]

\[
\text{Def } f3 \equiv fc3 \circ [id \ newstack]
\]

Once again we have exposed all the recursive calls in a tail-recursive position. This allows us replace the recursive calls with while loops. In this case, even though we could substitute to obtain a single function, things are a bit clearer for human readers if we leave it in two pieces:

\[
\text{Def } fc4 \circ [x \ c] \equiv f_{send} \circ [a \circ x \ c] \circ
\]

\[
\text{while not } p \circ x \ [l \circ x \ push \circ [[T \ x] \ c]]
\]

\[
\text{Def } f_{send} \circ [v \ c] \equiv (\text{null} \circ c - v;
\]

\[
fc4 \circ [r \circ val \circ c \ push \circ [[F \ v] \ pop \circ c]) \circ
\]

\[
\text{while not} \circ (null \circ c - T; \ tag \circ c)
\]

\[
[b \circ [val \circ c \ v] \ pop \circ c]
\]

\[
\text{Def } f4 \equiv fc4 \circ [id \ newstack]
\]

This is fairly comprehensible. The function fc4 sends the result of applying a to the leftmost leaf of its argument x after stacking all the right subtrees with the tag T to form the continua-
tion for that result. The function \texttt{fsend} restarts this process after pushing onto the stack the result of combining that leaf value with the value of all other previously evaluated left sub-trees which have been stacked with the tag \texttt{P}; if the stack is empty \texttt{fsend} merely returns the result it was passed.

5.3 Recursion removal algorithm

We now abstract from the above examples a general method for removal of recursion from functional programs. The main idea behind this approach is to move recursive calls to a tail-recursive position, whereupon the function may be converted to a \texttt{while} loop. In translating the continuation version of the application to a functional program representation with interpretation, we could merely apply the algorithm for the continuation of a functional form as given in Chapter 4, and then translate to a suitable representation. In the case of the self-referential functional forms, we will see that the atom used to represent the continuation may be pushed back onto the stack when the self-referential continuation would re-invoke itself. This leads us to a more clever implementation of a functional representation, as is given in the table below. We translate the original function \( f \) to a new version \( f' \) defined in terms of a continuation version \( fc \), along with a standard sending function:

\[
\text{Def } f' \equiv \text{fc} \circ [\text{id newstack}]
\]

\[
\text{Def } fsend \circ [v \ c] \equiv \text{null} \circ c - v; \ finterp \circ [v \ \text{tag} \circ c \ \text{val} \circ c \ \text{pop} \circ c]
\]

Thus the sending function merely returns its value parameter if there is no built-up continuation to use; otherwise it passes that value along with the dissected pair which was on the top of the stack, and with the rest of the stack for later reference. We introduce an interpretive function which uses those values whose general scheme will be:
Def \( \text{finterp} \circ [v \ r \ x \ c] \equiv eq \circ [r \ \bar{\rho}_1] - \cdots ; \)

\[ \cdots \]

\[ eq \circ [r \ \bar{\rho}_n] - \cdots ; \]

In this table, the form to be translated is given together with its translated version; in the case where a continuation token is pushed onto the stack, the clause used by the interpretive function (i.e., \( eq \circ [r \ \rho \bar{bar}] - \cdots \)) is given as well. The first version of the Insert form assumes \( \bar{e} \) to be the constant function whose value is the right unit \( e \) for \( f \); the second version assumes no such right unit.
Continuation Translation
Interpretive clause

\[ g: x \{ R \} \quad fsend \circ [g \circ x \ c] \]

\[ g: f:x \{ R \} \quad fc \circ \{ x \ push \circ [\overline{p} \ x] \ c \} \quad fsend \circ [g \circ v \ c] \]

\[ \{ f \ g \}: x \{ R \} \quad fc \circ \{ x \ push \circ [\overline{p} \ x] \ c \} \quad fsend \circ [v \ g \circ x \ c] \]

\[ (p-f; g): x \{ R \} \quad p \circ x - fc \circ [x \ c] \quad fsend \circ [g \circ x \ c] \]

\[ (f-g; h): x \{ R \} \quad fc \circ \{ x \ push \circ [\overline{p} \ x] \ c \} \quad v - fsend \circ [g \circ x \ c]; \quad fsend \circ [h \circ x \ c] \]

\[ (\overline{v}): x \{ R \} \quad fsend \circ [\overline{v} \ c] \]

\[ (b u f \ y): x \{ R \} \quad fc \circ [[\overline{y} \ x] \ c] \]

\[ i f: x \{ R \} \quad fsend \circ [eappend \circ \{ rev \circ distl \circ [\overline{p} \ x] \ c \} \quad fc \circ [[x \ \overline{v}] \ c] \]

or

\[ fsend \circ [\overline{p} \ append \circ \{ rev \circ distl \circ [\overline{p} \ rtl \circ x] \ c \} \quad fc \circ [[x \ \overline{v}] \ c] \]

\[ (a f): x \{ R \} \quad fsend \circ [\overline{p} \ append \circ \{ distl \circ [\overline{p} \ x] \ c \} \quad \overline{p}_1 - fc \circ \{ x \ push \circ [[\overline{p}_2 \ y] \ c] \}; \quad \overline{p}_2 - fsend \circ [apndr \circ [x \ \overline{v}] \ c] \]

\[ (w hile \ p \ f): x \{ R \} \quad fsend \circ \{ x \ push \circ [[\overline{p} \ x] \ c] \quad p \circ v - fsend \circ [v \ c]; \quad fc \circ [v \ push \circ [[\overline{p} \ v] \ c] \]

In the vector functional forms Insert and Applytoall we have stacked all at once everything we
need to do.
5.4 Tree insert, lookup example

The following Rosetta Smalltalk [Warr79] code was taken from an example suggested to prove the usefulness of functional programming in a simple "real-world" example. There is a nominal difference in notation due to conformance with Smalltalk syntax. In particular, "=>" is used in the place of "\---", with @RHO1 and @RHO2 denoting Smalltalk atoms used to implement Backus world atoms.

" Backus tree insert and lookup routines ".
" Scott Marks May 1980 ".

Def insert =
  (nulltree =>
    {empty item empty};
  isroot = =>
    {insert a [item left a tree] root a tree right a tree};
  isgtroot = =>
    {left a tree root a tree insert a [item right a tree]};
      tree)

Def lookup =
  (nulltree =>
    Constant F;
  istroot = =>
    lookup a [item left a tree];
  isgtroot = =>
    lookup a [item right a tree];
    Constant T)

Def nulltree = {isnull a tree}

Def istroot = {islt a [item root a tree]}

Def isgtroot = {isseq a [item root a tree]}

Def empty = (Constant phi)

" A single tree node is a sequence of three elements: ".
"   {<left subtree> <datum for this node> <right subtree> }".

Def left = first! Def root = second! Def right = third!
Representation of continuations in functional programs

The arguments of lookup and insert are pairs:

\[ (\langle \text{item} \rangle, \langle \text{tree} \rangle) \]

\begin{verbatim}
Def item = first
Def tree = second
\end{verbatim}

We can easily modify the lookup function to be non-recursive since all of its recursive calls already occur at the tail:

\begin{verbatim}
Def lookup2 =
  (not null tree a
   While (null tree => Constant F; or a listroot listroot)
   (listroot => (item left a tree; (item right a tree)))
\end{verbatim}

The insert function does have embedded recursive calls, so we use the techniques of the previous section:
Def insertc =
   (nulltree o arg =>
    insertsend o [[empty item empty] o arg cont]
   ) inserto arg =>
   inserto [[item left o tree] o arg push o [[Rho1 arg] cont]]
inserto arg =>
   inserto [[item right o tree] o arg push o [[Rho2 arg] cont]]
insertsend o [tree o arg cont]]
Def arg = -rst1 Def cont = second1
Def rho1 = (Constant @RHO1) Def rho2 = (Constant @RHO2)

Def insertsend = (null o cont => result;
insertinterp o [result tag o cont val o cont pop o cont]]
Def result = -rst1

Def insertinterp = (null o c => v;
   eq o [r rho1] => insertsend o
   [v root o tree o x right o tree o x] c);
   eq o [r rho2] => insertsend o
   [left o tree o x root o tree o x v] c);
Constant bottom)
Def v = -rst1 Def r = second1 Def x = third1 Def c = fourth1

Def insert2 = (insertc o [id newstack])]

Note that the absence of the FAD naming function convention has required us to generate separate naming functions for the parameters of each major program part. The next transformation to while loops to eliminate the recursion in insertc is omitted. We note that once the proper place for insertion is found, the stack represents a continuation which reconstructs the modified tree from the new node and the pieces on the stack. This action is conditioned only by whether the subtree is grafted on the left or right, and hence that function may be changed to an Insert form.

5.5 Mutually recursive functions

So far, we have dealt exclusively with single self-recursive functions. Functional programs offer the advantages of referential transparency, allowing the definition of a function to be substituted for any occurrence of its name. This technique is critical to procedure integration optimizations, and is simplified by the absence of parameter variables requiring renaming or
substitution. This advantage may be extended to systems of multiple mutually-recursive functions by a simple technique analogous to Gaussian elimination. We can phrase the method inductively:

Basis step:

In a system with a single self-recursive function, use the above techniques to reduce the number of recursive functions from one to zero.

Inductive step:

In a system with \( n \) mutually-recursive functions, use the above techniques to render any one function non-self-recursive. Then substitute the body of the modified version of that function for all its occurrences; the result will be a system of \( n-1 \) mutually-recursive functions.

This algorithm is non-deterministic in that any of the \( n! \) possible orders of recursion elimination is equally suggested. Since the call matrix in a system of functions is likely to be sparse, several heuristics for choosing an order leading to less-complicated modified versions of the functions suggest themselves. Cliques (strongly connected subsets) in the call relation should be eliminated together; references to members of the clique by other functions suggest that the original definitions of those members be re-introduced, substituting calls to the non-recursive versions in place of the original invocations. The structure of the call graph can be simplified by the techniques discussed in [Stro75] in which the number of strongly connected components is increased, thereby decreasing the number of functions within each component. In particular, the semigroup word problem considered there is even more likely to indicate a solution to the identification problem than in the context presented in that paper, since functional programs are exactly strings of procedure calls, the simplification which led to the algebraic formulation they considered. We might also analyze the functions from simplest
Representation of continuations in functional programs

to most complex, if a suitable concrete realization of that notion can be chosen.
6. Message-passing implementations of functional programs

Functional programs can express computations in a form which maximizes possible concurrency of execution by minimizing specification of the sequential order of computational events. This chapter discusses implementations of functional programs in computational systems which seek to exploit this possible concurrency. The implementations we examine center around a computational paradigm in which an argument is sent to a function representation, which in turn sends its result to further such representatives for use as their arguments. We examine a syntactic model for this paradigm similar to the $\lambda$-calculus, called the $\mu$-calculus. This model naturally suggests implementations of functional programs in networks of interconnected processors without being concerned with the specific details of a particular machine.

6.1 Message-passing and continuations

In the previous chapter we examined functional programs with the intent to analyze the history of each portion of the computation. We exerted much effort to using this analysis to remove recursion from recursively-defined functional programs. We will now see that this is only one aspect of the use of continuations. Recall that we chose the name "sending functions" for the introduced function which implements the action of applying the continuation which has been represented as a data structure to the result obtained by an application. This name was chosen to suggest the computational model we now investigate, in which we consider arguments as messages sent between functional objects.

6.1.1 Actors

Carl Hewitt and others propose a uniform model of computation called "actor systems" composed of "actors" which bundle the concept of data along with the procedural specifications of operations on that data. Each actor "acquaints" itself with those data items (primitive or
low-level actors) which form its local storage, as well as other actors which perform subroutine-like service. It understands a set of parametrized messages (carried by message actors) which request it to perform calculations based on its acquaintances in accordance with scripts ("methods") which define its characteristic behavior. If this sounds rather vague, this feeling is due not only to the unusual suggestive notation, but also to the desire to describe singularly primitive computational ideas.

We can see the similarity of the actor formalism to the modules of Simula and the packages of Ada. A distinction remains in that both the Simula and Ada conceptualizations are those of a combination of operations which manipulate a given data type with lower-level data items which form a representation of instances of that type. In particular, it is assumed that there are primitive data types which are essentially passive objects operated upon by primitive operations.

The actor formalism entails no such machinery-inspired ideas. Even the most primitive actors, such as integers, are active computational entities which respond to the messages sent to them. These primitive actors define themselves only in terms of their responses. Higher-level actors can modify their "internal state" by changing their set of acquaintances as a result of the receipt of some message, but only the actors themselves know the means they choose to implement this state. Thus we define an actor's "data type" only by the messages to which it responds and the patterns of behavior elicited by those communications. Special actors serve as message carriers to mediate the interactions between actors. Message receivers commonly respond by creating new message carriers, to return a response to the sender for example, or to cause further computation by sending messages on to other actors. Conventionally there are one or more continuation actor acquaintances packaged in these "envelope" actors along with the form of the message and the other acquaintances of the message carrier which form the parameters of the message. The receiver will be expected to forward its response to
such a continuation in accordance with its script. We would commonly find a normal ("then-


to") continuation along with an error ("complaint-dept") continuation known to each message,


which would then introduce the receiver to these acquaintances for its use.

6.1.2 Smalltalk

Several languages have been implemented using the actor formalism as a basis. PLANNER was Hewitt’s original robot control language. The PLASMA system has been

implemented at the MIT Artificial Intelligence Laboratory based on the actor transmission

communication mechanism using primitive actors coded in LISP. In conjunction with others

at the Xerox Palo Alto Research center, Hewitt developed the Xerox Smalltalk family of

languages. Several implementations of Rosetta Smalltalk, a language similar to Xerox’s

Smalltalk-72, were constructed by Scott Warren and Dennis Abbe at Rosetta, Inc. in Houston

for a variety of machines, including a prototype version for Z80-based microcomputer systems

which was used for the examples in this work. We now discuss the terminology and proper-

ties of the Smalltalk language in greater detail.

These languages share the central metaphor of actors as "objects" grouped into "classes"

which share common scripts. Each particular object is merely an instance of the class of

which it is a member. We distinguish among the classes by the messages which all members

of a class answer, and the shared methods all members use to answer those messages. Each

Class itself is an object acquainted with and known by all its instances, and as such pro-

vides a central location for data accessible by all members of the class, but unavailable to

objects of different classes. The class to which an object belongs also defines the set of names

it uses for its acquaintances. In general, different classes employ different methods for calcul-

ating responses to messages received. These differences induce the differences in behaviors

which are the actual distinguishing characteristic object of distinct classes.
If we choose to ignore the behavior which results from message receipt, and concentrate instead on the set of messages which a group of objects answer, we find that we can divide all objects into equivalence classes according to the messages answered, wherein the instances of more than one Smalltalk Class will be grouped together. We can consider this to be an equivalence relation "modulo messages answered". Even when two objects of different classes do not answer all the same messages, they may share the ability to response to some of the same message patterns. If the behavior elicited in the two objects is sufficiently similar (a subjective judgement), we can overlook the differences in a way analogous to generic functions in languages like PL/I. This so-called "polymorphism" promotes modularity by encouraging designers of a new clase to define answers to many of the same messages other classes answer in a way appropriate to the new class. For example, almost every object answers a message "print" by behavior which includes making a representation of itself visible to the Smalltalk system user on his interactive terminal. Each class of object used to implement some form of number will answer messages of the form "+ <another number of that type>" by sending to some receiver some object it chooses to represent the sum of itself with the other number.

In Smalltalk-72 and Rosetta Smalltalk the answer to a message is always returned to the sender, who is known to that implicit continuation known to all objects, the run-time stack. Thus these languages commit the "tail-recursion error" of unlimited stack build-up which prohibits extensive use of the "continuation-passing" style of iteration.

6.1.3 Control structures

In actor systems which do implement tail-recursion correctly, it is possible to form arbitrary control structures along the lines of the unrestricted flowchart ("goto") class of programs. Even without this feature, all the other common forms of structured programming control and data organization found in a language like Algol 60 can easily be implemented as message
sendings. An important occurrence of this is in the procedure-like objects. We mentioned above that most objects answer the message "print"; for instance, the message "2 print" will result in a "2" being printed somewhere on the terminal screen. This polymorphism allows the definition of a simple procedure-like object, typically called "print", which receives messages of the form "<something>" (that is to say, a message of the form "print <something>" is processed by the implementation) and in turn sends the message "print" to the object it was passed. This one method for "print" will allow this utility object to print a representation of any object which knows how to print itself; most importantly, the exact nature of that representation and how to print it remains the responsibility of the class of which the object is an instance, retaining locality of that information while allowing central access to an important concept.

Such a utility object is often the only member of its class. The class is often anonymous, with the user directly acquainted with only with the instance, which can then control his access to the class itself. Another use for utility objects is the instantiation of data storage, where in another language we would find a global data structure. These data would be accessed by sending messages to the utility object, which would then respond with that portion of its information it chose to reveal. In a similar way, a utility object can embody a collection of methods (pieces of program text) of general usefulness. One of the operations specified by those methods may be invoked within a chosen context by sending the appropriate message to the utility object. It is similar to a group of procedures when used in this way.

Common control structures are implemented by utility objects which receive an uninterpreted piece of program text as a message parameter. The receiver is free to evaluate the piece of code in a way which emulates the desired control pattern. For instance, loop controls are implemented by a collection of loop procedures. The "repeat" object receives a message consisting only of an code segment, and repeatedly evaluates it until a special "done" object
receives a message; the interpreter then pulls a bit of magic, terminating the loop and continuing interpretation at the next message. The "for" object recognizes a more complex message pattern similar in form to an Algol for statement. The message parameters include an unevaluated atom which is to be bound as the "controlled variable" along with iteration limits and the code to be executed. It would also be possible to have a utility object "if" which would behave in a manner analogous to an Algol if statement. The basic conditional execution pattern is achieved with a primitive conditional message sending construct, another bit of magic built in to the language.

6.2 $\mu$-calculus

Ward and Halstead have abstracted from the message-sending languages a syntactic model for message passing, the $\mu$-calculus, which is intended to be similar in form and function to the $\lambda$-calculus. The latter provides a syntactic model for applicative semantics, with the substitution of result for application corresponding to the call and return paradigm of functions. Message sending semantics divide this dyadic framework by concentrating on each half of the call-return activity as an instance of simple sending of messages under the assumption that the context originating the call message will be the receiver of the returned value. The $\mu$-calculus provides a strictly syntactic, axiomatic model for message sending. The introduction of the concept of conduits allows representation of interaction between streams of message sendings while retaining the determinancy and formal simplicity of applicative semantics.

6.2.1 Objects and events

The primitive notions of the $\mu$-calculus model the concepts of primitive actors and of message sendings in general. Objects and events are the abstract models for these concepts.

Objects are primarily members of a set of distinguished constants. These correspond to natural numbers, booleans and similar atomic objects; they form the data of the system, but
are somewhat removed from the concept of primitive actors in that they do not directly receive or send messages. These constants appear primarily inside message sending constructs which do behave as primitive actors. Also considered as objects are the variables which appear as parameters in messages. This grouping is justified in that the variables often name constants of the primitive data type, and thanks to referential transparency may be manipulated as if they did so even when they are actually naming more complex constructs.

We model message sendings by tuples of objects, which we call events. Although this is formally sufficient, we have a further intuitive layer of meaning placed on top by our desire to model messages. It is important to note that this intuitive meaning finds expression only in our interpretation of what is going on, and in the conventions we use in employing the axioms which describe the behavior of the system. We intend the first member of the tuple to be the receiver of the message, just as in Smalltalk code vectors in which the result of evaluating the first member of a vector is the object to which the remainder of the vector is sent as a message. Unlike Smalltalk, there are no unevaluated tokens providing syntactic sugar in this abstract model. Thus all the objects in the tuple are parameters of the message. In the versions of Smalltalk discussed above, we mentioned that the continuation which receives the answer to a message sending is always the implicit continuation denoted by the run-time stack, which we see as the sending context. In the $\mu$-calculus we wish to take a more PLASMA-like view of things with explicit continuations present, so conventionally the last member of a tuple will be an object which will be treated as the continuation.

Along with the primitive constants, the $\mu$-calculus semantics includes a set of primitive operations, which model addition, comparison, and so forth. Since we are attempting to model primitive message sending, we will typically choose primitive operations which act in a continuation-passing manner, like the ++ operation of Steele. This intention is seen in the causality axiom discussed below.
6.2.2 $\mu$-expressions and message sending

The $\lambda$-calculus provides $\lambda$-expressions as a means of abstracting particular applications into patterns of application. When such an expression is bound through an enclosing application to a parameter name, that name then denotes the pattern of application, and a substitution rule allows the same pattern of application to be employed wherever the name appears. The $\mu$-calculus contains a similar mechanism for abstraction of message sending patterns. $\mu$-expressions denote message patterns, and when a $\mu$-expression occurs in the receiver position of an event, the remaining objects in the event are bound to the parameters of the message pattern in a manner analogous to $\lambda$-calculus application. Primitive operations operate in a similar manner; the body in which substitution occurs is merely concealed, and the events which result may not correspond to those available within the system. The important distinction between the $\lambda$-calculus and the $\mu$-calculus is that the former specifies a process of reduction in which an application containing abstractions is replaced by the result of its calculation in the expression which encloses it, whereas the latter specifies a process of causation in which a message sending containing message patterns causes further messages external to it to occur; no replacement of result for invocation is intended. Following Ward and Halstead, we write $E \rightarrow F$ to mean "the event $E$ directly causes the event $F$". The first axiom of causality specifies the result of an event in which a $\mu$-expression appears as the receiver.

**Axiom 1.** If $E$ is the event $((\mu \ x_1 \cdots \ x_n \ . \ E_1 \cdots E_m) \ A_1 \cdots A_n)$ then $E \rightarrow F$ for every event $F$ of the form of some $E_i$ wherein each $A_j$ has been simultaneously substituted for $x_i$, where $i$ assumes each value from 1 to $m$.

We note some differences from the $\lambda$-calculus. We allow multiple parameters to $\mu$-expressions, where some $\lambda$-calculus systems require that all $\lambda$-expressions have a single parameter, with multiple-parameter applications being Curried. A non-divergent $\lambda$-calculus application specifies an expression to be substituted for the application, and thus has a single
expression as its body; an event with a $\mu$-expression as its receiver may cause more than one other event to occur, since no substitution is implied, and thus more than one message pattern may occur in the body of a $\mu$-expression. The second causality axiom specifies the result of an event in which a primitive operation appears as the receiver.

Axiom 2. If $E$ is the event $(p\text{SUB}1 \cdots \text{SUB}nR)$ where $p$ is a primitive action, and each $c_i$ is a constant, then $E\leftarrow (R \; f_1 \cdots c_{\text{SUB}n})$ where we have used $f$ to denote an $n$-ary function corresponding to the operation $p$.

Events involving primitive operations behave in a manner similar to Steele's $++$ operation, as noted earlier. Each primitive operation, $++$ for example, corresponds to an applicative function, $+$ in this case. The event $(++34R)$ causes the event $(R7)$, where the 7 is derived as the result of the application $+\cdots<34>$. An important point here is that the transformation of the application $+\cdots<34>$ to the constant $7$ occurs outside of the $\mu$-calculus; it is not intended that we think of the event $(R7)$ replacing the event $(++34R)$, but rather that the latter causes the former to occur.

We should note the continuation-passing nature of the original event, where a continuation for the off-stage result is included as a parameter in the message to the continuation version of the addition operation, and appears as the receiver of the event containing the result. We should also observe that the only events involving primitive operations contain only constants as the message objects; thus we do not pass primitive operations $\mu$-expressions, which does not allow them to inject events into the $\mu$-calculus system in a way not justified by the controlling axioms. Although message patterns involving primitive operations and $\mu$-expression parameters may occur within the body of a $\mu$-expression, by the time the associated event occurs, those parameters will have constants substituted for them if the primitive operation event is to be allowed by the axiom above.
6.2.3 Conduits

The "pure $\mu$-calculus" described above allows a single message event to cause a number of events, each of which can itself cause further events. This allows a forest of computational activity to be initiated from a collection of root events. It must be emphasized that the forest structure is in the mind of the beholder, for although an event can cause other events to occur, there is no other connection between events. In particular, we have no mechanism in which the two events can interact to cause a third which depends on both. If we are to use the $\mu$-calculus to model the kind of parallel processing we wish to use to implement functional programs, we need to be able to collect independently derived results for further processing. Ward and Halstead introduce the concept of conduits to play the role of joins for the forked computations caused by chains of $\mu$-expression events. Syntactically, we define a conduit as an ordered pair $<r_X w_X>$ of elements from a new class of objects distinct from constants, variables, primitive and $\mu$-expressions, but labelled with some $X$, a $\mu$-calculus object. For intuitive suggestion we call $r_X$ the read side of the conduit and $w_X$ the write side of the conduit, and in fact never denote the conduit object itself, but instead direct our attention to its components separately. Ward and Halstead use a distinct notation for the "long-range" causality attributed to activity involving conduits, called "ultimately causes", which aids in understanding proofs involving such events. We gloss over that difference in the following axioms.

Axiom 3. If $E$ is an event such that $E \rightarrow (r_X A)$ and $E \rightarrow (w_X B)$, then $E \rightarrow (A B)$.

Here we see that conduits capture the essence of the joining of two event sequences. The object $A$ sent to the read side of the conduit must be a $\mu$-expression or a primitive object for the resulting event $(AB)$ to cause any further events. The object $B$ sent to the write side of the conduit is the single parameter in the message $(AB)$ ultimately caused by the two earlier events. In $\lambda$-calculus discussions, the restriction to a single parameter, and resultant use of
Currying for multiple-parameter functions, leads to simplified proofs involving the subsequent syntactic manipulations. The extension to multiple parameter functions can be handled by the Currying operation in a consistent manner, and often in the literature multiple-parameter \( \lambda \)-expressions are allowed once the completeness and universality proofs are out of the way. For similar reasons, Ward and Halstead have restricted the write-side event to have a single object parameter; if multiple-parameter effects are needed, the parameters may be bundled in a \( \mu \)-expression in a manner exactly analogous to Currying with \( \lambda \)-expression parameters in \( \lambda \)-calculus functions. Having noted this, we will extend the above axiom to allow pairs of events of the form \((r_x A)\) and \((w_y B_1 \cdots B_n)\) to ultimately cause events of the form \((A B_1 \cdots B_n)\) without further notice.

Conduits are created by means of a new kind of event using the operator \( \xi \) and a particular object which labels the resulting conduit.

**Axiom 4.** The event \((\xi X)\) directly causes the event \((X r_x w_x)\), where \(X\) is a \( \mu \)-calculus object.

We observe that the object \(X\) must be a continuation for the read and write sides of the new-created conduit. The nature of the \( \mu \)-calculus enforces the uniqueness of the event \(X\) so far as a context for use of the conduit is concerned, and together with the fact that these axioms apply only to events, and not to event patterns buried within other \( \mu \)-calculus objects, it can be shown that this scheme is sufficient to identify conduits uniquely. In fact, it will usually be unnecessary to refer to the object \(X\) at all, and we will occasionally distinguish conduits in contexts in which more than one conduit is accessible by means of subscripts, which can be taken as abbreviations for the creation continuations.

We regard conduits in one light as a means of storage of information, in that if one side of the conduit has received a message, whatever information is embedded in that message
remains waiting for the other side’s event. According to the axioms above, we can visualize conduits as a pair of sets of the messages sent to each side of the conduit, enhancing the image of information storage event further. For instance, if a constant C has been sent to the write side of some conduit, it can be considered as a closure, since if the read side is then sent a μ-expression in an event like \((r \ (\mu x. (A (\mu y. y x))))\), this pair of events will cause the continuation \(A\) to receive the μ-expression \((\mu y. y C)\), which is simply the closure of the inner μ-expression above with respect to the substitution of \(C\) for the parameter \(x\). As we have seen in the previous chapter, data structures for closures are merely a matter of interpretation, such as this one.

Another aspect of conduits is as a means of value fanout. Suppose \(X\) is an event of the form \((\mu tw. (r \ (\mu x. ++ x x R)) (rR) (w3) (w4) (w5))\). The event \((\xi X)\) will then immediately cause the event \((X r_x w_x)\), by Axiom 4; μ-reduction as specified in Axiom 1 will then cause the five events

\[
(r_x (\mu x. ++ x x R))
\]

\[
(r_x R)
\]

\[
(w_x 3)
\]

\[
(w_x 4)
\]

\[
(w_x 5)
\]

These events will ultimately cause the six events

\[
((\mu x. ++ x x R) 3)
\]

\[
((\mu x. ++ x x R) 4)
\]

\[
((\mu x. ++ x x R) 5)
\]
(R 3)

(R 4)

(R 5)

The first three of those events will in turn cause

(R 6)

(R 8)

(R 10)

under the assumption that ++ is the continuation version of +, so that (++44R) will cause (R 8), as discussed above.

Another use for conduits will be seen below in the implementation of functional programs in μ-calculus forms. There we will use conduits as modular connector analogues, wherein the implementation of a function will create a write side of one conduit to which arguments will be sent and the read side of another conduit from which results can be obtained by sending that read side a μ-expression in an event like (r_{ans} (μy.Ry)) where r_{ans} is the answer conduit read side and R is the continuation for the answer. We will assert that the implementation for the function is correct if sending an argument to the argument conduit write side will ultimately cause the result to be sent to the result conduit write side.

6.2.4 Proof techniques

The μ-calculus defines a causal relationship between events. For instance, we say that the event ((μx.Rx) 3) directly causes the event (R 3) as a result of Axiom 1 of μ-reducibility. We extend this concept to the reflexive, transitive closure of the causality relation, saying for example that the event ((με.ε4) (μx.Sx)) causes the event (S 4) in two steps. We frame all the axioms and theorems of the pure μ-calculus in this way, asserting that any event which
satisfies some pre-condition causes some set of consequent events which satisfy some post-condition. The pure \( \mu \)-calculus provides no way for consequent events to depend on the conjunction of more than one event. Even so, we extend the notion of causality in the pure \( \mu \)-calculus to assertions that some set of events satisfying a pre-condition causes some set of events satisfying a post-condition. When conduits are introduced in the full \( \mu \)-calculus, consequent events do depend on more than one event, and in the full \( \mu \)-calculus this is the main framework for proofs. For instance, if \( \langle r w \rangle \) is a conduit, we say the two events \( (w (\mu x . T x)) \) and \( (r (\mu c . e 5)) \) ultimately cause the event \( (T5) \). Although Ward and Halstead found it helpful to distinguish the relations of direct causality within the pure \( \mu \)-calculus, causality (the reflexive, transitive closure of direct causality) and long-range or ultimate causality (of events involving conduits) for the purposes of proof, we will not make this distinction.

An interesting and useful way to look at the causality relationship takes note of the fact that nowhere in the \( \mu \)-calculus is the order of events specified. One intuitively assumes that an event which causes another must precede it temporally. However, the \( \mu \)-calculus accurately reflects the relativistic dilemma inherent in parallel execution, in which order of events is a only a partial order, and time is best modelled as a directed acyclic graph, rather than a linear path. Attempting to include such a time scheme unnecessarily complicates the picture, as all we are initially concerned with is the presence, in the set of events caused by an initial (input) event satisfying an input predicate, of a terminal (output) event which satisfies an output predicate. No one attempting to optimize calculations should object if a program produces results before using its input so long as the results are correct!

Thus for the purposes of correctness, we can consider a computation as merely a set of events. We specify the set constructively by asserting that the set contains some initial set of input events, and only those other events which are caused by events already known (recur-
sively) to be elements of the set. A theorem asserting the correctness of a μ-calculus program then merely says that if the computation contains a legal input event, then it must also contain a correct output event, and must not contain incorrect output events. This timeless way of looking at the computation is related to the intermittent assertion method of program correctness proof for conventional programs, in which the correctness lemmas assert that if control ever reaches point A in the program with a given predicate P satisfied, then control must sometime reach point B in the program with another given predicate Q satisfied. The correctness theorem for a program as a whole merely asserts that if control is ever at the program beginning with the input predicate satisfied, then control will sometime reach the end of the program with the output predicate satisfied, combining partial correctness and termination proofs into one theorem.

6.3 Implementations of functional programs

The μ-calculus provides a syntactic model for computations viewed as patterns of message-passing. We have investigated this model because we believe that it strongly suggests a mechanical implementation which makes good use of systems of multiple processors. The μ-expression component of the model seems to be a good analogue to the concept of λ-expressions as defining notational boundaries between the smallest pieces of computation, and the conduit component provides a good syntactic realization of the concept of the future of a datum, such as when a result is a parameter of a message to a continuation.

6.3.1 Translation to μ-calculus

In the μ-calculus, calculations are sets of events linked by causality. After each primitive event, other events are caused only by reduction of μ-expressions which caused the primitive event. Thus for chains of calculation to exist, events must include as parameters or else calculate some other way the consequent events which continue the calculation. Thus as our first step in translation of functional programs to μ-calculus, we transform the functions to
continuation-passing style to elicit the nature of the connections between the component calculations of the programs.

Recursive function invocations will result in the allocation of conduits which assume the role of argument copies saved during recursive evaluation. As we will see, some of this work will typically be optimized out in the conduit propagation optimization. However, the remainder must be seen as a dynamic allocation of processors to sub-tasks, incurring a log-squared penalty in more realistic ultracomputer-type models. Thus the extensive analysis in the previous chapter can be used to eliminate non-tail-recursive forms; it will be seen that the final step of conversion to a while loop can be seen either as a transformation within the functional programming system or as a conduit elimination after moving a constant operation from run-time to compile-time; the resulting structures are identical.

Tail-recursions, and tail-invocations in general, can make use of the message-passing version of the standard tail-recursion optimization. In the μ-calculus this means that the continuation passed to the next invocation can be calculated in the same small constant time as if it were to return, but that the message which would actually return the result to the invoking context before passing it in turn back to the invoking context of that sender can be optimized out, saving constant effort on each branch in the tree of recursive computation.

The translation from functional programs to μ-calculus must specify the representation within the μ-calculus world of both functions and objects, including the special object ⊥. The objects present the simpler considerations. First, the objects of the functional program system can represent themselves as the constants which were unspecified in the discussion of μ-calculus in general, and we can consider them as specifying a particular μ-calculus for which all results concerning the μ-calculus in general obtain. In a system with true actor semantics, there are no such passive objects as these; their role is take by the primitive actors, whose behavior follows unspecified and sometimes magic scripts, but which are still message
receivers like all others. For instance, the actor 2 will generally send itself as an acquaintance of the message which results from its receipt of a message, as in the event \texttt{2eval} requesting that it evaluate itself. Thus there is no passive "number 2" but instead only an actor 2 which knows about evaluation, arithmetic, printing, etc. We will ultimately translate functional programs to \( \mu \)-calculus objects which never deal with constants (FP objects) as such, but instead with their actor forms. Thus in the place of the sequence \( \langle A B \rangle \) we will in general find the \( \mu \)-expression \( (\mu c . c \langle A B \rangle) \). We can think of this as a call-by-name mechanism, which allows us to delay evaluation until an argument is needed. This alternate representation of FP objects will give rise to several representations for functions and continuations as discussed below.

We will represent functions as message-senders as in the discussion of ++ above. Each primitive function from Backus world will be translated to a primitive operation within the \( \mu \)-calculus. Thus where we have a primitive function \( f \) in Backus world, we have a primitive operation \( \texttt{f}\texttt{p} \) in the \( \mu \)-calculus, which we could define in a mixed notation as \( \texttt{f}\texttt{p} = (\mu x . c \ . f\texttt{x}) \). This is of course an illegal \( \mu \)-calculus expression, as we have no notion of function application within the \( \mu \)-calculus world. What we have done here is made use of a different notation which we consider "magic" with respect to the \( \mu \)-calculus as a means to specify what the primitive operation is to accomplish. It should be noted that for these to be primitive operations, the \( \mu \)-calculus requires that the parameters other than the continuation to be used be constants, which here means FP objects. Thus these versions of the primitive functions will use the representation of objects as themselves, not in the actor formulation discussed above. We can use this formulation in a different representation \( \texttt{f}\texttt{a} \) for the primitive function \( f \) in \( \texttt{f}\texttt{a} = (\mu a . a \ (\mu x . \texttt{fpx} (\mu y . (\mu d . d) y) c)) \). Note that if \( \texttt{TWOPAIR} \) is the actor version of the FP object \( \langle 22 \rangle \) as in \( \texttt{TWOPAIR} = (\mu c . c \langle 22 \rangle) \), and if \( +p \) is the \( \mu \)-calculus primitive operation corresponding to + (called ++ above), and if \( +a \) then is defined
like \( sa \) above: \( +a=(\mu a \cdot a (\mu x . +p x (\mu y . (\mu d . d y) c))) \), then for any continuation object \( R \), the event \( (+aTWOPAIR R) \) will cause the event \( ((\mu d . d 4) R) \), which is merely a renaming of the event \( (FOUR R) \), where \( FOUR \) is the actor version of the FP object 4: \( FOUR=(\mu c . c 4) \). Notice that the event \( (FOUR R) \) directly causes the event \( (R 4) \), the same event caused by the event \( (+p 2 2 R) \), and thus the two versions can be considered to be different only in their "functionality" with respect to argument type; we will use the term message type to denote this property of \( \mu \)-calculus objects similar to function type. We could have chosen instead to have actor versions cause the event (continuation result) directly by means of definitions like \( fa=(\mu a \cdot a (\mu x . fp x (\mu y . c y))) \). This formulation would not fit so well with the introduction of conduits as handled below; since the difference consists merely of replacing the event \( ((\mu d . d y) c) \) by its immediate consequent event \( (c y) \), we can think of this as a type of constant propagation to be performed after the functional program implementation has been defined. We should not necessarily think of the actor version of a function as being "more primitive" than the low-level (see below) version, as we can define \( fp=\mu x c . fa (\mu d . d x) c \) which reverses the roles as given above.

We have not explicitly noted the translations for \( \bot \), the special functional programming system object we use to represent illegal or divergent computations. This object could be implemented in the same way we did above, with \( \bot \) as a \( \mu \)-calculus constant with the actor version \( \bot =((\mu c . c \bot) \). This introduces a bit of inconsistency in that if the \( \mu \)-calculus representation of a function is sent an illegal argument it will send \( \bot \) to its continuation, but if it is sent an argument on which the function diverges, it will never send its continuation any result at all. Thus we choose simply not to represent the object \( \bot \) within the \( \mu \)-calculus, which is equivalent to choosing as a representation the behavior of the representation of a divergent application: "sending the continuation the value \( \bot \)" is represented by not sending the continuation any value. For example, if \( m0d \) is the primitive operation corresponding to
the FP modulus function, then the event \((\text{modp} <10\ 0> \ R)\) will result in no further events. This allows us a unified representation for error and non-termination analogous to the role of the object \(1\) within functional programming systems.

Our translation will follow the structure of the function being translated. The representation for a function defined as a functional form will be specified in terms of a \(\mu\)-expression composed of the representations of the parameter functions and objects with other structure effecting the connection between these parts in the behavior of the whole.

6.3.2 Low-level messages

We have noted that some translations will deal with all FP objects in their actor form. This means that it will be possible to segregate out subcomputations consisting of sets of events which are related by causality, and whose effect with respect to other events in the computation can be adequately summarized as \((f a\ A\ R)\ causes\ (B\ R)\) where \(A\) and \(B\) are representations of the argument and result of \(f:x(R)\), the original application with its continuation. The objects \(A\) and \(B\) must be defined so that if for any object \(S\), \((A\ S)\ causes\ (S\ x)\), then for any object \(T\), \((B\ T)\ must\ cause\ (T\ f:x)\). Note that even here, when we specify the behavior characteristic of the actor formulations of \(x\) and \(f:x\), we must deal in terms of messages with constants as parameters. We use the term low-level for events such as these which involve actual constants (FP objects) as components.

Low-level messages occur most often in partial value computation, in which the continuation object must represent a closure used as the continuation of the original functional application. This type of low-level message is unavoidable because of the need to form \(\mu\)-expressions which are the actor formulations of results, as in the sequence resulting from the event representing \(+:<2\ 2>R):\n
\[
(+a\ \text{TWOPAIR}\ R) = \\
((\mu\ a\ c.\ a\ (\mu\ x.\ +p\ x\ (\mu\ y.\ (\mu\ d.\ d\ y)\ c))))
\]
\((\mu \ c \ . \ c <2 \ 2>) \ R\) –

\((\mu \ x \ . \ +p \ x \ (\mu \ y \ . \ (\mu \ d \ . \ d \ y) \ R))\) –

\((\mu \ x \ . \ +p \ x \ (\mu \ y \ . \ (\mu \ d \ . \ d \ y) \ R)) \ <2 \ 2\)) –  \text{ (low level)}

\(+p \ <2 \ 2> \ (\mu \ y \ . \ (\mu \ d \ . \ d \ y) \ R)\) –  \text{ (low level)}

\(((\mu \ y \ . \ (\mu \ d \ . \ d \ y) \ R) \ 4)\) –  \text{ (low level)}

\(((\mu \ d \ . \ d \ 4) \ R) = (\text{FOUR R})\)

One effect of low-level message sending is that a form of improper value may be sent by the system. This is true only in the limited sense that one translation allows parallel execution of the representation of construction functional forms. Here it might be possible that at some point in time during the execution of an implementation, if in the FP version some of the component functions produce results where others diverge, some of the component function representations might have actually sent component results where others merely have not sent their results at that point. An optimized version of the representation could even allow a selector function following such a construction to determine the component in question sent directly to the selector continuation, but if some other components were truly divergent, this would have the effect of having sent a proper sequence (a sequence with some component \(1\) but not considered equal to \(1\) itself) to the selector. This is an example of the kind of extension that we earlier mentioned that we would allow, since if the sequence builder and selector representations are not optimized out, the sequence builder will never receive all its components, and hence will never send a sequence to the selector (that is, will send the selector the representation of \(1\)), so that the correct result (no result at all) will be obtained.

6.3.3 Parallel message sending

We now consider some of the consequences of implementations using parallel message sending. One place where parallel messages should arise is in the construction of static sequences from component function application. By static sequences we mean those
sequences whose extent is known at compile-time. This type of calculation can easily be distributed among multiple processors to advantage. We qualify the suggestion to use distributed computation in this situation only to the extent that the benefit obtained by having the execution cost be the maximum instead of the sum of the component costs must not be outweighed by the cost of communicating the argument to the component functions and accumulating the results in a sequence.

The use of Applycall and Insert in dynamic situations, such as the matrix multiplication program, present several complicating factors. We use the adjective "dynamic" to refer to situations involving a sequence whose length is not known at compile-time. Once again, if the benefits of parallel execution outweigh communication costs, we must dynamically allocate processors for assignment to the component calculations. The communication costs are greater in a dynamic situation, so that it will more often be worthwhile to contain the calculations within a single processor. In these cases, we can convert Alpha and Insert to the equivalent recursive formulations to calculate the results within a constant number of processors.

To make these assertions more concrete, we must consider the issues of communication between processors and the interconnection schemes which implement this communication. The terminology we will use in this discussion is that of Schwartz.

The first model, called the paracomputer model, is of a sufficiently large number of processors with total interconnection. This model is necessarily unrealistic, as a sufficiently large number of processors for large problems may be such that no practical processor hardware could support the required number of connections to all the other processors. A related model, also due to Schwartz, is the ultracomputer model, in which there are still enough processors, but each is limited to a constant number of connections to other processors. The interconnection method considered by Schwartz in [Ultracomputers] is the perfect shuffle connection of [Batcher], which allows communication of an argument to n processors in a tree of
messages of height $O(\log n)$, and calculation of an arbitrary permutation of processors, such as is required in the general case for dynamic processor allocation, with a cost of $O(\log^2 n)$. The perfect shuffle interconnection pattern is that of the fast Fourier transform, which exhibits a kind of overall symmetry, but not precise similarity of pattern from one region to the next. Recently the interconnection pattern called cube-corner-connected has been suggested in [Cube Corner Connections] as an alternative which exhibits the same asymptotic communication cost, but has a more regular interconnection pattern, more suitable to VLSI layout.

Another way to look at the cost of message sending is that of serial simulation. This implementation shares a technique used by multiprocessing systems, in which events caused during the course of a computation are placed on an event list for processing. Here we imagine the event list initialized with the initial set of events, and processed by repeatedly removing an event from the list, handling that event by adding to the list all the events it causes. This raises several issues. If the list is managed in a FIFO manner by removing from the front of the list and adding on the end, then even divergent subcomputations will not cause starvation of other parallel computations. In a parallel implementation of this technique, we can remove from the list any event whenever a processor is freed by having completed causing all the events consequent to the process it handled. Since all events on the event list will be "ready to execute", the speedup from having multiple processors will be linear with the number of processors available. Note that this is because we have bundled the communication overhead into the implementation of the functions involved, and that the price of this apparent efficiency may include a hidden cost due to copying of arguments.

Nonetheless, this technique constrasts favorably with the so-called "data flow machines", in which a RAM contains a number of activation records of processes whose parallel execution is being simulated, with a processor which is allocated by scanning the list for a record for an unblocked process. The action of process blocking is replaced with the action of sending a
process continuation to the read side of a conduit, so that when the argument is sent to the write side, the computation will proceed, essentially scheduling itself.

Because of the $O(\log^2 n)$ cost of organizing distributed computation in a network of realizable processors, we see that there is a trade-off between the cost of performing a computation locally in a serial fashion in a constant number of processors versus the cost of distributing the computation completely among enough processors so that all component computations can proceed in parallel. This threshold will depend on a sensitive way on the precise cost of calculations and communication, attributes of the particular hardware used in an implementation, and not something we handle in a theoretical way here. It should also be noted that completely parallel computation will have a ceiling as well as its trade-off floor, because any real system will have a finite number of component processors, and thus will have a limit to the parallelism possible. This could also become a non-trivial limit if the system were capable of creating new processors on demand at some cost.

6.3.4 Translation algorithms

Having discussed some of the issues in implementation of functional programs by message-passing systems, we now proceed to exhibit several related translations of functional programs to $\mu$-expressions with the belief that the latter can be efficiently implemented.

One issue discussed above is that of the representation of the constants of the $\mu$-calculus, which we take to be the set of FP objects. We refer to the constants themselves as the I. "native form" of the objects. We use the term I. "actor form" to mean the objects bundled inside a $\mu$-expression which will pass the object to a continuation, or if we prefer we can think of them as the same objects translated into actor semantics.

Since functions map argument objects to result objects, and continuations map result objects to answers, the representation we choose for objects induces the correct message type
of the representation for functions and continuations. The native message-sending forms of
primitive functions and functional forms will take constants, the native form of objects, and a
continuation as parameters, and will send the native form of the result object to the continu-
ation as their final act. We assume that each primitive function is translated to just such a
primitive operation of the \( \mu \)-calculus. This is necessary without regard to other consider-
ations. If this form of translation is followed all the way through, it is necessary to explicitly
obtain arguments from objects which act like actor forms of constants. By this last phrase, we
mean objects of the form \((\mu e . <\text{something}>\)) where the events of the body will cause an
event of the form \((e <\text{constant}>\)). This would not seem much of a problem, but since we
would like to exploit the fanout characteristics of conduits, we find that this explicit dere-
ferencing is a problem since, as noted previously, the read side of a conduit acts like an actor
form for the objects sent to the associated conduit write side.

For the purposes of the \( \mu \)-calculus formalism we will assume the primitive operations to
be the native-mode representations of the primitive functions. We will immediately define
forms of those primitive functions which use actor forms for parameters, and can consider
those forms to be atomic since the low-level messages will be contained within these actor
formulations of primitive functions. This decision is effectively to use call-by-name rather
than call-by-value because of the namelike behavior of the read side of conduits. As noted
earlier, the question of what kind of continuation to use turns out to be unimportant, since
event though a native mode function sends a constant result to its continuation, an actor
mode form sends its result form the same continuation it was given, which will ultimately
cause a final result constant to be sent to the outermost continuation. This is a convenient
formalism, since we will be trying to prove correctness, and the assertion that if for all objects
R and S the objects A and B satisfy \((A R) - (R x)\) and \((B S) - (S f:x)\), then the implementa-
tion \(f a\) of a function \(f\) is correct if \((f a A R) - (B R)\), which is structurally similar to saying \(f\)
applied to $x$ is $f \cdot x$. Nonetheless, we can consider these representations to be of a half-native half-actor type, since a native-style continuation-gets-constant event is the ultimate result. In many cases, this formulation is more natural, and the translations given below will define such an event, only to bundle the constant back into an actor form for the sake of the formalism. Fortunately, the message-sending version of the procedure integration optimization can easily remove these redundancies. In the translations using conduits, we will use the write side of a conduit as the continuation for such a low-level event directly, safe that if we send the result continuation to the read side, the latter event has the correct message type for the formalism, while the implied events are essentially direct transfer of the low-level data.

We now present three translations which assume this actor-actor form of representation. In all that follows we will use the notation $\tau(x)$ for some $\mu$-calculus object $x$ to mean $(\mu \ c \ . \ c \ x)$, the actor form of that object. Since the object will often be a parameter, we need this notation to distinguish this concept from that of the translated form of a variable in an application, which will in general be the variable itself. We will be dealing with function representations as $\mu$-expressions all of the form $(\mu \ a \ . \ <\text{something}>)$ where, as noted previously, the $<\text{something}>$ will cause $(b \ d)$, with $b$ replaced by the actor form of the result, and $d$ replaced by the same continuation passed to the application originally. A shorthand notation will help us here also. We often will need to represent the result of substituting a specific continuation for the continuation parameter in a $\mu$-expression such as the one above, or in a native form representation as well. Note that the concepts of Currying a $\mu$-expression already of the correct form and of closing the simpler multiple-parameter $\mu$-expression with respect to one of its parameters are essentially identical. We copy the form of continuation application here: if $fp = (\mu \ x \ . \ c \ <\text{something}>)$, then $fp(R) = (\mu \ x \ . \ R \ x)$; if $fa = (\mu \ a \ . \ <\text{something}> d)$, then $fa(R) = (\mu \ a \ . \ <\text{something}> R)$. These are merely suggestions of what is actually a process of substitution for parameters, but this will often occur in situations like
those above. We find this notation convenient since here the combination of a function representation with a continuation is effectively just a modified continuation in the same way that functions and continuations work in the λ-calculus.

The first translation assigns to each functional program a μ-expression within the pure μ-calculus. Since the pure μ-calculus has no method for joining parallel paths of message sending, the opportunities for exploiting potential parallelism are lost as sequential execution is necessary. Nonetheless, the connection between the continuation forms of functional programs and the corresponding μ-expressions is enlightening. We translate to the actor form of function representation to provide a close connection with the next translation. In the following table, O[f] is the translated form of the function f. Each translated form is designed so that the event (O[f] C) results in the continuation C being sent the actor form representation of the function f. We mean this to indicate a "compile-time" message sending, where we are preparing to use the representation of f at a later time. In the case of a recursive function, this representation will also be sent to the write side of a conduit, so that the representation can refer to itself through the read side of that conduit.

<table>
<thead>
<tr>
<th>Functional form</th>
<th>Pure μ-calculus expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>μ c . O[g] (μ ga . O[f] (μ fa . c (μ a d . ga a (μ x . fa τ(x) d))))</td>
</tr>
<tr>
<td>Construction</td>
<td>μ c . O[f] (μ fa . O[g] (μ ga . c (μ a d . ga a (μ x . fa a (μ y . τ(&lt;x y&gt;) d))))</td>
</tr>
<tr>
<td>Condition</td>
<td>μ c . O[p] (μ pa . O[f] (μ fa . O[g] (μ ga . c (μ a d . pa a (μ t . (SEQ2 τ(t) a (μ q . (COND a τ(p) fa[d] ga[d]))))))</td>
</tr>
<tr>
<td>Constant y</td>
<td>μ c . c (μ a d . a (μ x . τ(y) d)) (non-strict)</td>
</tr>
<tr>
<td>bu f x</td>
<td>μ c . O[f] (μ fa . c (μ a d . a (μ y . fa τ(&lt;x y&gt;) d))))</td>
</tr>
</tbody>
</table>


Message-passing implementations of functional programs

\[
\text{Insert} \quad \mu \, c \cdot O[f] \ (\mu \, f \cdot a \cdot c \\
\quad (\mu \, a \cdot d \cdot a \cdot (\mu \, x \cdot \tau (\langle x \cdot e \rangle) \ S))) \\
S = \mu \ y \cdot \text{firstp} \ y \ (\mu \ z \cdot \text{nullp} \ z \ (\mu \ t \cdot \ \text{SEQ2} \ \tau(t) \ \tau(y)) \ (\mu \ q \cdot \ \text{CONDa} \ \tau(q) \ (\mu \ y \cdot \text{secondp} \ y \ d) \ \ (\mu \ y \cdot \text{firstp} \ y \ (\mu \ z \cdot \text{rdhp} \ z \ (\mu \ x_1 \cdot \ \text{secondp} \ y \ (\mu \ x_2 \cdot \ f_a \ \tau(\langle x_1 \ x_2 \rangle) \ (\mu \ z \cdot \text{firstp} \ y \ (\mu \ x \cdot \ \text{rlp} \ x \ (\mu \ x \cdot \ \tau(\langle x \ z \rangle) \ S)))))))))) \]
\]
\(\text{(c is the right unit for f)}\)
\[
\alpha \ f \quad \mu \ c \cdot O[f] \ (\mu \ f \cdot a \cdot c \\
\quad (\mu \ a \cdot d \cdot a \cdot (\mu \ x \cdot \tau (\langle x \cdot \phi \rangle) \ S))) \\
S = \mu \ y \cdot \text{firstp} \ y \ (\mu \ z \cdot \text{nullp} \ z \ (\mu \ t \cdot \ \text{SEQ2} \ \tau(t) \ \tau(y)) \ (\mu \ q \cdot \ \text{CONDa} \ \tau(q) \ (\mu \ y \cdot \text{secondp} \ y \ d) \ \ (\mu \ y \cdot \text{firstp} \ y \ (\mu \ z \cdot \text{hdhp} \ z \ (\mu \ z \cdot \ f_a \ \tau(z) \ (\mu \ z \cdot \text{firstp} \ y \ (\mu \ x \cdot \ \text{rlp} \ x \ (\mu \ x \cdot \ \text{secondp} \ y \ (\mu \ x \cdot \ \text{apndp} \ (\langle x \ z \rangle) \ S)))))))))) \\
\]
\[
\text{while p f} \quad \mu \ c \cdot O[p] \ (\mu \ p \cdot a \cdot O[f] \ (\mu \ f \cdot a \cdot c \ (\mu \ a \cdot d \cdot a \ S)) \\
S = \mu \ y \cdot \text{pa} \ \tau(y) \\
\quad (\mu \ t \cdot \text{SEQ2} \ \tau(t) \ \tau(y)) \ (\mu \ q \cdot \ \text{CONDa} \ \tau(q) \ f_a[S] \ d) \]
\]

The self-referential nature of the Insert, Apply to all and While functional form representations results in the necessity to name their continuations so as to let those continuations refer to themselves. Nonetheless, as we did in the last chapter, we can consider this to be an abbreviation for application of the paradoxical combinator. In this case we must use the translation of the Y-operator to \(\mu\)-calculus as suggested by Ward and Halstead.

We recognize the highly nested nature of the above translations, and feel that some suggestions for reading them may help them seem more transparent. Comparison with the translations of functional forms to continuations in the last chapter will show these new translations to be merely the \(\mu\)-calculus form of the same analysis. In each case, the compile-time continuation is sent the actor form of the function representation in a message of the form \(c \ ((\mu \ a \ d \cdot \langle \text{something} \rangle))\). This event is in turn caused in a continuation event by some number of previous events which cause the translation of the parameter functions of the
functional form to be sent to the continuation event. The primitive functions which appeared in the translations in the last chapter are represented here by the corresponding primitive operations, which have been consistently labelled with the name of the primitive function followed by a the letter p. For instance, secondp is a $\mu$-calculus primitive operation which has the property that any object S the event (secondp x S) will cause the event (S second:x) exactly when x is a sequence of more than one component. As noted, in the case that x is not such an FP object, so that second:x = L, the event (secondp x S) will simply cause no further events. We do not define the actor form of conditional CONDa, but merely specify its behavior. Assuming that B is the actor form of <p x>, with C and D some other objects, then the event (CONDa B C D) will cause ($\tau$(x) C) if p=T (that is, if (B R) causes (R T)), will cause ($\tau$(x) D) if p=F, and will cause no further event otherwise.

The next translation exploits conduits in strategic places to rejoin split paths of message sending which implement parallel evaluation of functional forms. Here we see the utility of using the actor-actor formulation when it is quite easy to replace parameters with conduit read sides.

This translation assigns to each functional program a $\mu$-expression within the full $\mu$-calculus. Since the full $\mu$-calculus has a method for joining parallel paths of message sending, the opportunities for exploiting potential parallelism are regained as sequential execution is unnecessary.

<table>
<thead>
<tr>
<th>Functional form</th>
<th>Full $\mu$-calculus expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition f o g</td>
<td>$\mu$ c. O[g] (\mu g\ a\ .\ \ O[f] \ (\mu\ f\ a\ .\ c \ (\mu\ a\ d\ .\ \xi\ (\mu\ r\ w\ .\ ((g\ a\ w)\ (f\ a\ d))))))</td>
</tr>
<tr>
<td>Construction if g</td>
<td>$\mu$ c. O[f] (\mu f\ a\ .\ \ O[g] \ (\mu g\ a\ .\ c \ (\mu a d . \xi \ (\mu r w . ((f a w) \ (g a (\mu y . SEQ2 r \tau(y) d)))))</td>
</tr>
<tr>
<td>Condition p-f\ g</td>
<td>$\mu$ c. O[p] (\mu p\ a\ .\ \ O[f] \ (\mu f\ a\ .\ O[g] \ (\mu g\ a\ .\ c \ (\mu a d . \xi \ (\mu r w . ((f a w) \ (g a (\mu y . SEQ2 r \tau(y) d)))))</td>
</tr>
</tbody>
</table>
Message-passing implementations of functional programs

\[(\mu \cdot (\mu \cdot (\mu \cdot x \mapsto (a \cdot (x \mapsto \mu r. (\mu t. (\pi2 r. (\mu q. (\text{CONDA} r. (\mu a \cdot d. \cdot a. (\mu x. \cdot r. d.)))))))))))(\mu c. \cdot \xi. (\mu \cdot r. w. (\mu \cdot w. x. (\text{O} f. (\mu f. a. c. (\mu \cdot a. d. \cdot a. (\mu y. \cdot \text{SEQ2} r. \tau. y. (\mu z. \cdot f. a. \tau. z. d.)))))))))\]

\[
\begin{align*}
\text{Constant } y & \quad (\mu c. \cdot \xi. (\mu \cdot r. w. (\mu \cdot w. y. (c. (\mu \cdot a. d. \cdot a. (\mu x. \cdot r. d.)))))) \quad \text{(strict)} \\
& \quad \text{or-} \\
& \quad (\mu c. \cdot \xi. (\mu \cdot r. w. (\mu \cdot w. y. (c. (\mu \cdot a. d. \cdot r. d.))))) \quad \text{(non-strict)}
\end{align*}
\]

\[
\begin{align*}
\text{buff} & \quad (\mu c. \cdot \xi. (\mu \cdot r. w. (\mu \cdot w. x. (\text{O} f. (\mu f. a. c. (\mu \cdot a. d. \cdot a. (\mu y. \cdot \text{SEQ2} r. \tau. y. (\mu z. \cdot f. a. \tau. z. d.)))))))))
\end{align*}
\]

\[
\begin{align*}
\text{Insert} & \quad (\mu c. \cdot \text{O} f. (\mu f. a. \cdot \xi. (\mu \cdot r. w. (\mu \cdot w. y. (c. (\mu \cdot a. d. \cdot a. (\mu x. \cdot r. f. (\mu c. \cdot c. \cdot w.)))))))))
\end{align*}
\]

\[
\begin{align*}
\text{If} & \quad (r. \mu y. \cdot \text{firstp} y. (\mu z. \cdot \text{nullp} z. (\mu t. \cdot \text{SEQ2} r. \tau. y. \cdot (\mu q. \cdot \text{CONDA} r. (\mu y. \cdot \text{secondp} y. d.))) - (\mu y. \cdot \text{firstp} y. (\mu z. \cdot \text{rightp} z. (\mu x_1. \cdot \text{secondp} y. (\mu x_2. \cdot f. a. \tau. ((<x_1 x_2>)))))))))))
\end{align*}
\]

(c is the right unit for \(f\))

\[
\begin{align*}
\text{αf} & \quad (\mu c. \cdot \text{O} f. (\mu f. a. \cdot \xi. (\mu \cdot r. w. (c. (\mu \cdot a. d. \cdot a. (\mu x. \cdot r. f. (\mu c. \cdot c. \cdot w.)))))))))
\end{align*}
\]

\[
\begin{align*}
\text{while pf} & \quad (\mu c. \cdot \text{O} f. (\mu f. a. c. (\mu a. d. \cdot \xi. (\mu \cdot r. w. (\mu \cdot a. y. \cdot p. a. \tau. y. (\mu t. \cdot \text{SEQ2} r. \tau. y. (\mu q. \cdot \text{CONDA} r. (\mu y. \cdot f. a. \tau. w. d.))))))))
\end{align*}
\]

Here we handle the self-referential nature of the Insert, Applytoall and While functional
form representations by using conduits to implement the continuations. This can be most easily seen in the while translation. The compile-time continuation is sent the usual (μ a d . <something>) form. In this formulation, the body of this μ-expression consists of two events. When the representation of the while is used, the actor form of the argument is merely sent to the write side of the conduit, while the read side of the conduit is sent the μ-expression which does the work of testing the argument. If that μ-expression, through CONDa, causes its "true" continuation to be sent the "x-value" actor form, the result of applying f to that argument (sending fa the actor form of the argument) will be to send the result (f:x in the FP world) to the write side of the conduit as a second argument to be processed.

In the Constant and bu functional translations, we use the conduit as a storage device for the object parameter. Here the conduit is instantiated at compile time when the object parameter is sent to its write side. Thereafter, the read side of the conduit exhibits the same behavior as an actor form of the constant without the implied copying of the call-by-value semantics otherwise used.

The Construction functional translation used the conduit in a minimal manner to allow the computations of f and g to proceed in parallel. The future for the value f:x is provided by the write side of the conduit; the future for the value g:x is a μ-expression which sends that result to a pair-building object along with the read side of the conduit in an event which will result in the original continuation being sent the pair of results bundled into a sequence. For a construction of n components, we could allocate n-1 conduits, with the n-1 write sides being the futures for the first n-1 result components, and the future of the last result component being a μ-expression which would cause all the read sides to receive messages which would in turn result in the final result being sent to the original result continuation.

We use the same primitive operations as in the last translation. In addition we have used SEQ2, one of a family of related objects for which we do not give further definitions. The
behavior we expect for SEQn is that an event of the form

$$(\text{SEQn } A_1 \ldots A_n \ R)$$

will cause the event

$$\tau(\langle x_1 \ldots x_n \rangle \ R)$$

if each $A_i$ is the actor form of the corresponding component $x_i$.

We now present the last translation, which is merely a modification of the second. In this translation we ensure that each function uniformly provides its argument actor with a conduit write side for its immediate continuation, and sends the application continuation to a conduit read side through which it will obtain the constant (FP object) result. We also extend the above translation by assuming the ability to allocate conduits whose number is not determined until run-time to handling sequences in the Insert and Applytoall forms. In those translations the number $n$ will refer to the length of the sequence argument.

<table>
<thead>
<tr>
<th>Functional form</th>
<th>Total conduit $\mu$-calculus expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition $f \circ g$</td>
<td>$\mu \ a \ d \ . \ \xi \ (\mu \ \text{rx wx.)}$</td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rf wf.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rg wg.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rc wc.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{(a w.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{fa rx wf.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{ga rx wg.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{SEQ2 if rg wc})$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{rc d)})$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Construction $[fg]$</th>
<th>$\mu \ a \ d \ . \ \xi \ (\mu \ \text{rx wx.)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\xi \ (\mu \ \text{rf wf.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rg wg.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rc wc.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{(a w.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{fa rx wf.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{ga rx wg.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{SEQ2 if rg wc})$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\text{rc d)})$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition $(p-fg)$</th>
<th>$\mu \ c \ . \ O[p] \ (\mu \ \text{pa . O[f]}) \ (\mu \ \text{fa . O[g]}) \ (\mu \ \text{ga . c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu \ a \ d \ . \ \xi \ (\mu \ \text{rx wx.)}$</td>
<td></td>
</tr>
<tr>
<td>$\xi \ (\mu \ \text{rq wg.)}$</td>
<td></td>
</tr>
</tbody>
</table>
\( \xi \ (\mu \ \text{rf w}f. \ 0) \\
\xi \ (\mu \ \text{rg w}g. \ 0) \\
\xi \ (\mu \ \text{rc w}c. \ ((a \ \text{wx}) \\
\quad (\text{rx} \ (\mu \ x. \ \text{pa} \ \tau (x) \ (\mu \ t. \\
\quad \text{SEQ2} \ \tau (x) \ \tau (t) \ \text{wq}))) \\
\quad (\text{CONDa} \ \text{rq w}f \ \text{wg} \\
\quad (\text{fa} \ \text{rf w}c \\
\quad (\text{ga} \ \text{rg w}c \\
\quad (\text{rc d}))))))))) \\
\text{Constant } y \quad \mu \ c . \ \xi \ (\mu \ \text{ry w}y. \\
\quad ((\text{wy y}) \\
\quad (c \ (\mu \ a \ d . \\
\quad \xi \ (\mu \ \text{rx w}x. \\
\quad \xi \ (\mu \ \text{rc w}c. \\
\quad ((a \ \text{wx}) \\
\quad (\text{rx} \ (\mu \ x . \ \text{ry w}c)) \\
\quad (\text{rc d})))))))) \\
\text{or-} \\
\mu \ c . \ \xi \ (\mu \ \text{rw} . \\
\quad ((\text{wy y}) \\
\quad (c \ (\mu \ a \ d . \ r \ d)))) \\
\text{lnu } f x \\
\mu \ c . \ \xi \ (\mu \ \text{rx w}x . \\
\quad ((\text{wx} x) \\
\quad (O[f] \ (\mu \ \text{fa} . \ c \\
\quad (\mu \ a . \ \xi \ (\mu \ \text{ry w}y. \\
\quad \xi \ (\mu \ \text{rf w}f. \\
\quad \xi \ (\mu \ \text{rc w}c. \\
\quad ((a \ \text{wy}) \\
\quad (\text{SEQ2} \ \text{rx} \ \text{ry w}f) \\
\quad (\text{fn} \ \text{rf w}c) \\
\quad (\text{rc d})))))))))) \\
\text{Insert} \\
\mu \ c . \ O[f] \ (\mu \ \text{fa} . \ n \ \xi \ (\mu \ \text{r}z_{1}, \ \text{w}z_{1}. \\
\quad \xi \ (\mu \ \text{r}z_{2}, \ \text{w}z_{2}. \\
\quad \xi \ (\mu \ \text{r}f_{i}, \ \text{w}f_{i}. \\
\quad \xi \ (\mu \ \text{rx w}x. \\
\quad \xi \ (\mu \ \text{rc w}c. \\
\quad ((a \ \text{wx}) \\
\quad (\text{firsta} \ \text{rx w}z_{1}) \\
\quad \ldots \\
\quad (\text{ntha} \ \text{rx w}z_{n-1}) \\
\quad (\tau (e) \ \text{w}z_{n-1}^{2}) \\
\quad (\text{SEQ2} \ \text{r}z_{n-1}^{2}, \ \text{r}z_{n-1}^{2} \ \text{w}f_{n-1}) \\
\quad (\text{fa} \ \text{rf}_{n-1} \ \text{w}z_{n-1}^{2}))
Message-passing implementations of functional programs

\[(\text{SEQ} \ r_1 \ r_2 \ \text{SEQ} \ r_1 \ r_2 \ w_{n-2} \ w_{n-2})\]
\[(\text{fa} \ r_{n-2} \ w_{n-2})\]

\[...\]
\[(\text{SEQ} \ r_1 \ r_2 \ w_f)\]
\[(\text{fa} \ r_f \ w_{r_f})\]
\[(r_1 \ \text{wc})\]
\[(r_1 \ d))]]]] \ ... \ ]]]]]]

(e is the right unit for \(f\))

\[\alpha \ f \quad \mu \ c \ . \ O[l] \ (\mu \ f a \ . \ \xi \ \xi (\mu \ r x \ w x) \]
\[\xi (\mu \ r x) \ w x_i\]
\[\xi (\mu \ r) \ w y_i\]
\[\xi (\mu \ r) \ w c\]
\[(\text{fa} \ w x)\]
\[(\text{firsta} \ r x \ w x_i)\]

\[\ldots\]
\[(\text{nth} \ r x \ w x_n)\]
\[(\text{fa} \ r x_1 \ w y_1)\]

\[\ldots\]
\[(\text{fa} \ r x_n \ w y_n)\]
\[(\text{SEQ} \ r y_1 \ \ldots \ r y_n \ w c)\]
\[(r_1 \ d))]]]] \ ... \ ]]]]]]

while \(p \ f\)
\[\mu \ c \ . \ O[p] \ (\mu \ p a \ . \ O[l] \ (\mu \ f a \ . \]
\[c (\mu \ o \ d) \ . \ \xi \ (\mu \ r x) \ w x)\]
\[\xi (\mu \ r x) \ w q\]
\[\xi (\mu \ r f) \ w f\]
\[\xi (\mu \ r c) \ w c\]
\[(\text{fa} \ w x)\]
\[(\text{rx} \ (\mu \ x) \ . \ p a \ t(x) \ (\mu \ t) \]
\[(\text{SEQ} \ t(x) \ t(\tau) \ w q)))\]
\[(\text{COND} \ a \ r x \ w f \ w c)\]
\[(\text{fa} \ r f \ w x)\]
\[(r_1 \ d))]]]]]]]]]]]]]

Here we again handle the self-referential nature of the Insert, Applytoall and While functional form representations by using conduits to implement the continuations. This can be most easily seen in the While translation. The compile-time continuation is sent the usual \(\mu\) form. a d . <something> form. When this \(\mu\)-expression is used, four conduits are instantiated. One conduit transmits the arguments, as in all the translations. The actor formulation of the predicate function receives its argument through this conduit and constructs the actor form of a pair with its truth value and the original argument which it sends to its conduit. The
conditional object CONDa is given that conduit read side as its input actors, and will cause the event \((\tau(x) \text{ wf})\) if the predicate is true, or \((\tau(x) \text{ wc})\) if it is false. The latter event just transmits the argument to the result continuation, corresponding to the identity function nature of a while functional with false predicate. If the predicate is true, the event \((\tau(x) \text{ wf})\) passes the argument to the "body of the while" function \(f\), which uses the argument conduit read side as the future for its value as in the preceding translation.

In the Insert and Applytoall functional translations we have attempted to maximize the parallel execution by allocating \(3n\) or \(2n\) argument-result conduits for any argument of length \(n\). The translation as written is therefore suspect since we allocate these conduits before having seen the argument! Nonetheless, we can consider this just shorthand for a more complex event which does examine the argument. Such an event could proceed as in the first translation, repeated allocating two conduits for each component of the argument as it appears at the head of the remaining part of the initial sequence. We would also need to have a couple of extra conduits to handle the boundary conditions.

The Insert functional translation as written is meant to imply that we set up three conduits for each component. The first element of the three is to receive on its write side the corresponding component of the argument. The second member of the triple receives on its write side the result of the Insert calculation on the tail of the argument sequence after the corresponding element. The read sides of the first two are sent to SEQ2 along with the write side of a third. The read side of the third is sent to \(fa\) as its argument actor, with \(fa\)-sending its result to the second write side of the next three.

The Applytoall functional works in a similar manner, with two conduits per component. One write side of the pair again receives each component, with \(fa\) using that read side as its argument actor and passing its result to the write side of the other member of the pair of conduits. The read sides of the second members of all the pairs are sent to SEQn in a big event
which gathers all the result components into the result sequence to be passed along to the initial continuation.

Where in the second translation we used the primitive operations corresponding to the primitive FP functions directly, here we are using conduits everywhere. We have therefore used the actor form of the primitive functions in this translation. For instance, seconda is a $\mu$-calculus primitive operation which has the property that if $A$ is the actor form of $x$, that is an object such that, for any object $R$, the event $(A \ R)$ will cause the event $(R \ X)$, then for any object $S$ the event (seconda $A$ $S$) will cause the event $(S \ second:x)$ exactly when $x$ is a sequence of more than one component. As we have mentioned previously, in the case that $x$ is not such an FP object, so that $second:x = 1$, the event (seconda $A$ $S$) will simply cause no further events.

The careful reader will discover that in several places in the last translation we have used more conduits than strictly necessary in order to make the presentation simpler. We use the term conduit propagation to denote the elimination of redundant conduits in a message-sending construct. We justify this transformation by the observation that if $<r_1 \ w_1>$ and $<r_2 \ w_2>$ are two conduits, and if the only event involving $r_1$ or $w_2$ is the event $(r_1 \ w_2)$, then that event can be eliminated if each event involving $r_2$ is replaced with the same event with $r_1$ substituted for $r_2$, on the basis that for any objects $A$ and $B$,

$$(w_1 \ A), (r_1 \ w_2) \text{ and } (r_2 \ B) \rightarrow (B \ A)$$

but we also have

$$(w_1 \ A) \text{ and } (r_1 \ B) \rightarrow (B \ A).$$

It would be difficult to see how the second set of three events would not be preferable to the first group of four events of similar complexity. We note that the last translation makes a the combination of the procedure integration and conduit propagation optimizations particularly effective.
The last translation as presented has the interesting property that all low-level events involve application of the conduit Axiom 3, rather than μ reduction. Thus we can imagine that conduits are implemented in a uniform manner to handle these low-level values. An associated effect is that we can look at the function translations as given in two ways. First, each application continuation is presented to the read side of a conduit in an event derived from (rc c) by substitution. This event is similar to the desired behavior of the actor form of a function in that the read side of a conduit with a constant on its write side behaves like the actor form of that constant. However, if the actual events caused are examined, we see that an event of the form (A R), where A is truly the actor form of a constant n, never occurs. Instead, the conduit events cause the result transmission event (R n) directly. We have observed that this is the ultimate result of the events involving the actor forms of functions as translated by the first translation as well. Nonetheless, we could consider the last translation as specifying μ-expressions of hybrid message type, which expect actor forms as arguments and send result constants to their continuations. The result of conduit propagation optimization is to transform all functions to that form, with the effect that the top-level semantics should be specified as

\[(\text{program argument continuation}) - (\text{continuation result})\]

rather than the form which we have been discussing

\[(\text{program argument continuation}) - (\text{result continuation})\]

This would require an outside layer for complete semantic cleanliness in which the original argument was a constant n, and the outside layer packaged it into τ(n) to conform with the message type of the representations of the function. We could consider this as well as part of the compile-time message sending.
Bibliography

Acke79  Ackerman, W.
Data Flow Languages

Alle72  Allen, Frances E. and Cocke, John
A Catalogue of Optimizing Transformations, in R. Rustin (ed.) Design and Optimiza-
tion of Compilers
Prentice-Hall, 1972

Alle76  Allen, Frances E.
Annotated Bibliography of Selected Papers on Program Optimization

Apt80  Apt, Krzysztof R., Francez, Nissim, and De Roever, Willem P.
A Proof System for Communicating Sequential Processes
ACM Transactions on Programming Languages and Systems 2, 3 (July 1980), 359-385.

Arvi81  Arvind and Kathail, Vinod
A Multiple Processor Data Flow Machine that Supports Generalized Procedures
News (SIGARCH Newsletter) 9, 3 (May 1981)

Ausl78  Auslander, M. A., and Strong, H. R.
Systematic Recursion Removal

Back78  Backus, John
Can Programming Be Liberated from the von Neumann Style? A Functional Style
and Its Algebra of Programs
Comm. ACM 21, 8 (August 1978), 613-641.

Batu68  Batcher, K. E.
Sorting networks and their applications
AFIPS SJCC 1968, pp. 307-314

Baud78  Baudet, Gerard and Stevenson, David
Optimal Sorting Algorithms for Parallel Computers
IEEE Transactions on Computers, C-27, 1 (January 1978) pp. 84-87

Bird77a  Bird, R. S.
Notes on Recursion Elimination

Bird77b  Bird, R. S.
Improving Programs by the Introduction of Recursion

Bird80  Bird, R. S.
Tabulation Techniques for Recursive Programs
Bibliography

Born79  Bornig, Alan
        ThingLab -- A Constraint-Oriented Simulation Laboratory
        Stanford Univ. Report STAN-CS-79-746 (July 1979)

Beye75  Boyer, Robert S. and Moore, J Strother
        Proving Theorems about LISP Functions

Burs75  Burstall, R. M. and Darlington, J.
        Some Transformations for Developing Recursive Programs
        Int. Conf. on Reliable Software, IEEE Computer Society, April 1975, 465-472.

Burt80  Burton, Richard R.; Masinter, L. M.; Bobrow, Daniel G.; Haugeland, Willie Sue;
        Kaplan, Ronald M. and Shell, B. A.
        Overview and Status of Dorado LISP

Cart80  Cartwright, Robert
        A Constructive Alternative to Axiomatic Data Type Definitions

Chea72  Cheatham, T. E. and Wegbreit, Ben
        A Laboratory for the Study of Automatic Programming

Darl73  Darlington, J. and Burstall, R. M.
        A system which automatically improves programs.

Darl76  Darlington, J., and Burstall, R. M.
        A System which Automatically Improves Programs

Deme79  Demers, A. and Donahue, J.
        Revised Report on Russell

Deme80a Demers, A.; Donahue, J. and Boehm, H.
        An Informal Description of Russell

Deme80b Demers, A. and Donahue, J.
        The Russell Semantics: An Exercise in Abstract Data Types

Denn74  Dennis, Jack B.
        FIT Version of a Data Flow Procedure Language

Deut80  Deutsch, L. Peter
        ByteLisp and its Alto Implementation

Down80  Downey, Peter J.; Sethi, Ravi and Tarjan, Robert Endre
        Variations on the Common Subexpression Problem
Earl74a  Earl, Jay
High Level Iterators and a Method for Automatically Designing Data Structure
Representation
Univ. of Cal., Berkeley Memo. No. ERL-M425 (February 1974)

Earl74b  Earl, Jay
High Level Operations in Automatic Programming
ACM SigPlan Symposium on Very High Level Languages. SigPlan Notices 9, 4 (April
1974), 34-42.

Earh74  Earnest, Christopher
Some Topics in Code Optimization

Basic Design of a Josephson Cache Memory
Yorktown Heights, New York

Fong75  Fong, Amelia; Kam, John and Ullman, Jeffrey
Application of Lattice Algebra to Loop Optimization
Conf. Rec., Second ACM Symposium on Principles of Programming Languages, Palo

Frie75  Friedman, Daniel P. and Wise, David S.
Cons Should Not Evaluate its Arguments

Frie76  Friedman, Daniel P. and Wise, David S.
Functional Combination

Fran80  Francez, Nissim
Distributed Termination
ACM Transactions on Programming Languages and Systems 2, 1 (January 1980), 42-
55.

Gerr75  Gerhart, Susan L.
Correctness Preserving Program Transformations
Conf. Rec., Second ACM Symposium on Principles of Programming Languages, Palo

Gesc72  Geschke, Charles M.
Global Program Optimizations
Ph. D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh,
Pa., 1972.

Goue77  Goguen, J. A.; Thatcher, J. W.; Wagner, E. G. and Wright, J. B.
Initial Algebra Semantics and Continuous Algebras

Gold76  Goldberg, Adele and Kay, Alan (eds.)
Smalltalk-72 Instruction Manual
Xerox Palo Alto Research Center SSL 76-6 (March 1976)

Gold80  Goldstein, Ira P. and Bobrow, Daniel
Extending Object Oriented Programming in Smalltalk
Bibliography


Gost79 Gostelow, Kim p. and Thomas, Robert E.
A view of dataflow

GreI75 Greif, Irene and Hewitt, Carl
Actor Semantics of PLANNER-73

GreI79 Greif, Irene and Meyer, A.
Specifying Programming Language Semantics: A Tutorial and Critique of a Paper by Hoare and Lauer

Guib78 Guibas, Leo J. and Wyatt, Douglas K.
Compilation and Delayed Evaluation in APL

Hend76 Henderson, Peter and Morris, Jr., James H.
A Lazy Evaluator

Hewi67 Hewitt, Carl
PLANNER: a Language for Proving Theorems in Robots
Mass. Inst. of Tech. (project MAC) AI Memo 137 (July 1967)

Hewi74 Hewitt, Carl, et al.
Behavioral semantics of non-recursive control structures

Hewi76 Hewitt, Carl
Viewing Control Structures as Patterns of Passing Messages

Hewi80 Hewitt, Carl
The Apiary Network Architecture for Knowledgeable Systems

Hibb80 Hibbard, Peter
Multiprocessor Software Design

Inga78 Ingalls, Daniel H. H.
The Smalltalk-76 Programming System Design and Implementation

Jone79 Jones, Neil D. and Muchnick, Steven S.
Flow Analysis and Optimization of LISP-like Structures
Bibliography

Kapl80  Kaplan, Marc A. and Ullman, Jeffrey D.
A Scheme for the Automatic Inference of Variable Types

Kild73  Kildall, Gary A.
A Unified Approach to Global Program Optimization

Knut74  Knuth, Donald E.
Structured Programming with go to Statements
Computing Surveys 6, 4 (December 1974)

Kosi76  Kosinski, Paul R.
Mathematical Semantics and Data Flow Programming

Kosi78  Kosinski, Paul R.
A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs

Lamp73  Lamport, Leslie
The Coordinate Method for the Parallel Execution of DO Loops

Lamp74a Lamport, Leslie
The Parallel Execution of DO Loops
Comm. ACM 17, 2 (February 1974), 83-93.

Lamp74b Lamport, Leslie
The Hyperplane Method for an Array Computer

Lamp75  Lamport, Leslie
Parallel Execution of Array and Vector Computers

Lieb80  Lieberman, Henry and Hewitt, Carl
A Session with TINKER: Interleaving Program Testing with Program Design

Love75  Loveman, David
Program Optimization, Theory and Practice

Love76  Loveman, David B.
Program Improvement by Source-to-Source Transformation.

Lowr80  Lowry, Edward S.
Nodes and Aces, the Ideal Primitive Data Structures
Bibliography

Mann73  Manna, Zohar, Ness, Stephen and Vuliemin, Jean.
Inductive Methods from Proving Properties of Programs

Mann77a Manna, Zohar and Waldinger, Richard
The Logic of Computer Programming

Mann77b Manna, Zohar and Waldinger, Richard
Synthesis: Dreams => Programs

Mann77c Manna, Zohar and Shamir, Adi
The Optimal Approach to Recursive Programs

Mann78a Manna, Zohar and Waldinger, Richard
Is "Sometime" Sometimes Better than "Always"?

Mann78b Manna, Zohar
Six Lectures on the Logic of Computer Programming

Marti80 Martin, Johannes J.
FAD, a Functional Programming Language that Supports Abstract Data Types

More79 Morel, E. and Renvoise, C.
Global Optimization by Suppression of Partial Redundancies
Comm. ACM 22, 2 (February 1979), pp. 96-103.

Morr77 Morris Jr., James H. and Wegbreit, Ben
Subgoal Induction

Mose80 Moses, Joel
The Function of FUNCTION in LISP

Much80 Muchnick, Steven S. and Pleban, Uwe F.
A Semantic Comparison of Lisp and Scheme

Myer80 Myers, G. J. and Buckingham, B. R. S.
A Hardware Implementation of Capability-based Addressing

Nass79 Nassimi, David and Sahni, Sartaj
Bitonic Sort on a Mesh-Connected Parallel Computer

New62 Newell, Allen
Some Problems in Basic Organization of Problem-Solving Programs
Rand Corp. Memo. RM-3283-PR (December 1962)

Plot78 Plotkin, G.
T-OMEGA as a Universal Domain
Bibliography


Prab80 Prabhala, Bhaskaram and Sethi, Ravi
Efficient Computation of Expressions with Common Subexpressions
J. ACM 27, 1 (January 1980), 146-163.

Pres75 Presburg, David L. and Johnson, Neil W.
The Paralyzer: IVTRAN's Parallelism Analyzer and Synthesizer
Proc of Conf. on Prog. Langs. and Compilers for Parallel and Vector Machines, ACM
SIGPLAN Notices 10, 3 (March 1975), pp. 9-16.

Raou80 Raoult, Jean-Claude and Vuillemin, Jean
Operational and Semantic Equivalence Between Recursive Programs

Ratl80 Rattner, Justin and Cox, George
Object-Based Computer Architecture
Computer Architecture News 8, 6 (October 1980), pp. 4-11.

Reyn70 Reynolds, John C.
GEDANKEN -- A Simple Typeless Language Based on the Principle of Completeness
and the Reference Concept

Reyn74a Reynolds, John C.
On the Relation Between Direct and Continuation Semantics

Reyn74b Reynolds, John C.
Towards a Theory of Type Structure

Reyn77 Reynolds, John C.
Semantics of the Domain of Flow Diagrams
J. ACM 24, 3 (July 1977), pp. 484-503.

Rite74 Ritchie, Dennis M. and Thompson, Ken
The UNIX Time-Sharing System
Comm. ACM 17, 7 (July 1974), 365-375.

Rose75 Rosen, Barry K.
How to Prelude Optimization
IBM Thomas J. Watson Research Center RC 5468 (1975)

Rose76a Rosen, Barry K.
The Toy Language Syndrome

Rose76b Rosen, Barry K.
Correctness of Parallel Programs: the Church-Rosser Approach
Bibliography

Rose77  Rosen, Barry K.
High-Level Data Flow Analysis

Rose78  Rosen, Barry K.
Monoids for Rapid Data Flow Analysis
Conf. Rec., Fifth ACM Symposium on Principles of Programming Languages, Tuscon,

Schw74  Schwartz, J. T.
Automatic and Semi-automatic optimization of SETL
ACM SigPlan Symposium on Very High Level Languages. SigPlan Notices 9, 4 (April
1974), 43-49.

Schw75  Schwartz, J T.
Automatic Data Structure Choice in a Language of Very High Level
Conf. Rec., Second ACM Symposium on Principles of Programming Languages, Palo

Schw80  Schwartz, J. T.
Ultracomputers

Scol72  Scott, Dana
Mathematical Concepts in Programming Language Semantics

Scol77  Scott, Dana S.
Logic and Programming Languages

Seth74  Sethi, Ravi
Testing for the Church-Rosser Property

Seth75  Sethi, Ravi
Complete Register Allocation Problems

Seth80  Sethi, Ravi and Tang, Adrian
Constructing Call-by-Value Continuation Semantics
J. ACM 27, 3 (July 1980), 580-597.

Shoc79  Shoch, John F.
An Overview of the Programming Language Smalltalk-72
SigPLAN Notices 14, 9 (September 1979), pp. 64-73.

Stee76a Steele, Guy Lewis Jr. and Sussman, Gerald Jay
LAMBDA: The Ultimate Imperative

Stee76b Steele, Guy Lewis Jr.
LAMBDA: The Ultimate Declarative

Stee77  Steele, Guy Lewis Jr.
Debunking the "Expensive Procedure Call" Myth

Stee78a Steele, Guy Lewis Jr. and Sussman, Gerald Jay
The Revised Report on SCHEME, a Dialect of Lisp
AI Memo No. 452, MIT AI Lab, January 1978.

Stee78b Steele, Guy Lewis Jr.
RABBIT: A Compiler for Scheme

Stee79 Steele, Guy Lewis Jr. and Sussman, Gerald Jay
Constraints
4-part 1 (June 1979), pp. 208-225.

Ston71 Stone, Harold S.
Parallel Processing with the Perfect Shuffle
IEEE Transactions on Computers, C-20, 2 (February 1971) pp. 153-161

Stoy77 Stoy, Joseph A.
Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory

Str71 Strong, H. R. Jr.
Translating Recursion Equations into Flow Charts

Str75 Strong, H. R. Jr.; Maggilio-Scheitini, A. and Rosen, B. K.
Recursion Structure Simplification

Tars55 Tarski, Alfred
A Lattice-Theoretical Fixpoint Theorem and Its Applications

Tart80 Tartar, John
Multiprocessor Hardware: an Architectural Overview

Tenn76 Tennent, R. D.
The Denotational Semantics of Programming Languages

Thom77 Thompson, C. D. and Kung, H. T.
Sorting on a Mesh-Connected Parallel Computer

Thor80 Thoreson, S. A. and Oldehoeft, A. E.
Instruction Reference Patterns in Data Flow Programs

Vuill74 Vuillemin, J.
Correct and Optimal Implementations of Recursion in a Simple Programming Language

Wand80a Wand, Mitchell
Continuation-Based Program Transformation Strategies
Bibliography

Wand80b Wand, Mitchell
Continuation-Based Multiprocessing

Ward79 Ward, Stephen A. and Terman, Christopher J.
An Approach to Personal Computing
Mass. Inst. of Tech. Lab. for Comp. Sci. (December 1979)

Ward80 Ward, Stephen A. and Halstead, Jr., Robert H.
A Syntactic Theory of Message Passing

Warr79 Warren, Scott K. and Abbe, Dennis
Rosetta Smalltalk: A Conversational, Extensible Microcomputer Language
Proc. of the Second Symp. on Small Systems, Dallas, Texas (October 1979), pp. 35-43.

Wats79 Watson, Ian and Gurd, John
A prototype data flow computer with token labelling

Wegb71 Wegbreit, Ben
The ECL Programming System

Wegb74a Wegbreit, Ben
Procedure Closure in EL1
The Computer Journal 17, 1 (February 1974).

Wegb74b Wegbreit, Ben
The Synthesis of Loop Predicates

Wegb75a Wegbreit, Ben
Goal-Directed Program Transformation
Xerox Palo Alto Research Center, September 1975.

Wegb75b Wegbreit, Ben
Mechanical Program Analysis
Comm. ACM 18, 9 (September 1975), 528-539.

Wegb75c Wegbreit, Ben
Property Extraction in Well-Founded Property Sets

Whit80 White, Jon L
Address/Memory Management for a Gigantic LISP Environment

Will80a Williams, John H.
On the Development of the Algebra of Functional Programs
IBM Technical Report draft, April 1980

Will80b Williams, John H.
On the Universality of Formal Functional Programs
IBM Technical Report draft, July 1980
Bibliography

Wino76  Winograd, Terry
Towards a Procedural Understanding of Semantics

Zahn74  Zahn, Charles T. Jr.
A Control Statement for Natural Top-Down Structured Programming