Combinator Graph Reduction:

A Congruence and its Applications

by

David Lester

A thesis submitted to the Faculty of Mathematical Sciences for the degree of Doctor of Philosophy, July 1988

Technical Monograph PRG-73
ISBN 0-902928-55-4
April 1989

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

ABSTRACT

David Lester
The King's Hall and College of Brasenose
and
Programming Research Group

A thesis submitted to the Faculty of Mathematics
for the degree of Doctor of Philosophy
July 1988

Combinator Graph Reduction: A Congruence and its Applications

The G-machine is an efficient implementation of lazy functional languages developed by Augustsson and Johnsson. This thesis may be read as a formal mathematical proof that the G-machine is correct with respect to a denotational semantic specification of a simple language. It also has more general implications. A simple lazy functional language is defined both denotationally and operationally; both are defined to handle erroneous results. The operational semantics models combinator graph reduction, and is based on reduction to weak head normal form. The two semantic definitions are shown to be congruent.

Because of error handling the language is not confluent. Complete strictness is shown to be a necessary and sufficient condition for changing lazy function calls to strict ones. As strictness analyses are usually used with confluent languages, methods are discussed to restore this property.

The operational semantic model uses indirection nodes to implement sharing. An alternative, which is without indirection nodes, is shown to be operationally equivalent for terminating programs.

The G-machine is shown to be a representation of the combinator graph reduction operational model. It may be represented by the composition of a small set of combinators which correspond to an abstract machine instruction set. Using a modified form of graph isomorphism, alternative sequences of instructions are shown to be isomorphic, and hence may be used interchangeably.

Contents

List of Figures

Acknowledgements

I would like to thank my supervisors, Phil Wadler and Richard Bird, for their encouragement. Phil, my supervisor from January 1985 to January 1988, agreed that a proof of the correctness of the G-machine would constitute a useful project in the world of functional programming. The use of equational reasoning in Chapter 8 was suggested by Phil. Richard then explained what equational reasoning was. Both spent considerable time checking the congruence proof of Chapter 3. Any remaining problems are purely my own work. The layout of the congruence proof of Chapter 3 was suggested by Michael Goldsmith.

Chapter 8 owes a lot to the paper presented at the third Functional Programming Languages and Computer Architecture conference. This benefited considerably from the helpful comments of Simon Peyton Jones and three other reviewers. The interest shown by attendees of FPCA '87 made the process of writing up easier.

The final months of writing up were made more comfortable by Geoff Burn, who made many useful observations on the work. He has been tireless in exposing points where the mathematics were not obvious. Thanks go to Juliet Shearman and Evelyn Lester for proof reading the completed work so promptly.

Finally to the Science and Engineering Research Council, without whom this work would not have been started; and GEC Hirst Research Centre, without whom this work would not have been completed so easily.

Chapter 1

Introduction

The simple denotational semantics of lazy functional programming languages allows powerful program transformation techniques to be used when developing programs. If the objective, when using these methods, is the production of a working system, then the implementation of the language should also be correct with respect to the denotational semantics. In this case the whole system may be said to meet its specification.

The G-machine is an efficient implementation of lazy functional languages which has been developed by Augustsson and Johnsson in a series of papers [1983, 1984], culminating in their theses [1987, 1987]. The work presented in this thesis may be read as a formal mathematical proof that the G-machine is correct with respect to a denotational semantic specification of a simple language. In fact the work is more general. Combinator graph reduction to weak head normal form will be shown to successfully implement lazy functional languages. This remains true even though the δ-rules that are selected in this thesis make the language non-confluent. The confluence of the pure λ-calculus is shown by the Church-Rosser Theorem, and for this reason confluence is often referred to as the Church-Rosser Property. Informally, it states that all reduction orders that terminate reach the same normal form.

This is important because most implementations of lazy functional programming languages are based on combinator graph reduction, where graphs are reduced to weak head normal form. Previously, results from the λ-calculus, such as The Church-Rosser Theorem, have been used in proofs about the operational properties of programs written in these languages. This thesis puts both the implementations and the program proof techniques

onto a firm mathematical foundation.

1.1 Functional Programming

The language examined in this thesis has its origin in notations developed for mathematical logic in the 1930's. These are: Church's λ-calculus [1936, 1941]; Kleene's general recursive equations [1936, 1950]; and combinatory logic, independently discovered by Schönfinkel [1924] and Curry [1930]. Each of these notations is at once both powerful and simple. For this reason, terms written in these notations are often easily shown to be equivalent. The problem with such powerful notations has always been efficient implementation.

The first practical realisation of these languages occurred in the early 1960's, when John McCarthy *et al.* developed LISP [1962]. Although this was based on the λ-calculus, it differed in two important respects. Firstly, the scope rules were dynamic rather than static. Secondly, the evaluation order chosen was applicative rather than normal. If a λ-term had a normal form then normal order reduction would find that normal form, see [Curry and Feys 58]. Unfortunately, applicative order reduction is less powerful, in the sense that it is not able to reduce all λ-terms with a normal form to that form. When this occurs the applicative order reduction fails to terminate. It therefore produces an approximation to the required result, rather than a contradictory one. In spite of the deviation from the pure λ-calculus, LISP is still popular.

During the early 1960's, Landin developed a virtual machine to implement statically scoped LISP. Because this machine used four registers; Stack, Environment, Code and Dump; it is called the SECD machine. It is described in [1964]. In fact, with a small modification this machine could be adapted to perform normal order reduction, but it would be hopelessly inefficient. The problem was that each use of an argument within a function required repeated evaluation of that argument.

Lazy evaluation is an attempt to overcome this difficulty. The result of evaluating an argument is preserved, so that subsequent uses of the argument benefit from the original evaluation. Landin's SECD machine may be made to perform this operation by using the concept of closures to represent unevaluated arguments. A closure is a pair consisting of some representation of the function and an environment where the function may access its local variables. This was developed by Henderson and Morris in the context

of data structures [1976]. A description of the general method occurs in [Henderson 80, Henderson *et al.* 82].

An alternative representation for closures was devised by Wadsworth [1971]. In place of the function and environment pair, a piece of graph is constructed. This graph represents the function body, with the variables of the environment bound into the correct places of the body. In Wadsworth's thesis [1971] graph reduction is used to evaluate the pure λ-calculus using reduction to head normal form. Turner [1979a, 1979b] showed how graph reduction could be used to evaluate combinatory logic terms efficiently. General recursive equations may be converted to pure combinators. Supercombinators [Hughes 82b] and lambda-lifting [Johnsson 83] are two methods described in the literature. It is programs that have been transformed to either of these forms, and now require reduction to weak head normal form, that we shall be concerned with in this thesis.

Graph reduction has proved to be important as it underlies the most efficient implementations of lazy functional languages known. Johnsson's G-machine was the first really efficient implementation based on graph reduction and providing lazy semantics. Further refinements to the graph reduction machine include Burn, Robson and Peyton Jones' Spineless G-machine [1988] and Fairburn and Wray's TIM [1987]. Another advantage of graph reduction is that it may easily be performed in parallel. This is because it has no global environment structure.

A number of introductory texts on functional programming exist. Burge [1975] describes recursive techniques in some detail, although he is not interested in reduction orders. Henderson [1980] describes LispKit, a simple language based on Lisp. It includes the development of the compiler, based on Landin's SECD machine [1964]. The techniques involved in developing algorithms written in lazy functional languages is described by Bird and Wadler [1988].

A good general introduction to the techniques of graph reduction occurs in Peyton Jones' textbook: "The Implementation of Functional Programming Languages" [1987].

1.2 Denotational Semantics

Denotational semantics is only one of a number of approaches to the specification of programming language semantics. It was developed at the Programming Research Group by Strachey [1966] and given a formal basis by

Scott's work on models for the λ-calculus [1970].

The approach taken in this thesis is that the semantics should be specified denotationally. This has advantages over the other two common specification methods. With an operational semantics we could arrange things so that no work is performed to establish whether graph reduction correctly implements our langnage. The problem is that the language is then over-specified. Other implementation methods would be precluded unless we could establish some form of operational equivalence between the states used in each implementation. A particular example in this context would be the SECD machine, suitably adapted to perform lazy evaluation.

An axiomatic semantics is usually provided for languages with state, and specifies a relationship between initial and final states, after executing a command. This immediately raises problems in a language without commands. Josephs [1986] has presented an axiomatic semantics for a lazy functional language with side effects. A similar axiomatic specification occurs in term rewriting systems. From an expression and a set of rewrite rules, we may deduce a new expression. The problem with this approach is that termination of the term rewriting system will normally be proven separately.

The most important advantage of denotational semantics for specifying lazy functional languages is that it is simple. This is not surprising; the λ-calculus is the basis of both denotational semantics and lazy functional languages.

The problem is then to show that graph reduction successfully implements the language specified. Of course we must also specify graph reduction. In this thesis we do so operationally, although an equivalent denotational version could be formulated. As the operational semantics of graph reduction provides a specification of an implementation it must contain more detail, and its semantics becomes correspondingly more complicated.

Although the notation used in this work is based on [Stoy 77], there are other texts which could be used as an introduction to this work. Milne and Strachey [1976] is a presentation of a congruence proof; this proof is used to motivate the investigation of denotational semantics. Gordon [1979] and Tennent [1983] are simple introductions, but perhaps too superficial. A recent textbook on denotational semantics, by Schmidt [1986], brings sections of Stoy's book up to date.

1.3 Overview

The work described in this thesis falls conveniently into two parts.

In Part I, a congruence result for a simple lazy functional languages is established. The syntax of this language is given in Figures 2.1 and 2.2. The remainder of Chapter 2 provides a denotational semantics and an operational semantics for this language. The denotational semantics is directly related to that for the λ-calculus given in [Stoy 77]. The operational semantics chosen is based on graph reduction. This model uses indirection nodes to implement sharing, but is otherwise similar to typical real implementations of graph reduction.

Chapter 3 proves that the two semantics given in Chapter 2 are congruent. The congruence of the two semantic definitions means that they agree upon the result of executing a program, whenever this is a base value. This relation is not equality as the results are drawn from different domains; the denotational semantics produces a result in the domain $\mathbf{E}$ where $\mathbf{E} = [\mathbf{E} \to \mathbf{E}] + \mathbf{B}$; the operational semantics produces an interpreter state. A further complication emerges because the operational semantics is able to equivalence more functions than the denotational semantics. This is the full abstraction problem, which we circumvent by considering congruence rather than equivalence.

The congruence proof is outlined in Figure 3.1 on page 24. It follows the classic pattern of such proofs, by establishing that each semantics approximates the other. Using Schmidt's terminology [1986], we may say that the operational semantics is both faithful and terminating with respect to the denotational semantics. Faithfulness is demonstrated by a fixpoint induction; the termination property by structural induction using inclusive predicates.

The language is extended in Chapter 4 to implement typical built-in functions such as addition, equality and data structuring functions. The operational semantics is extended so that there are no explicit recursive calls to the interpreter. The congruence result is then extended. This involves showing that the reduction steps associated with the built-in functions are faithful. Next we demonstrate that a new set of predicates, including one for comparing data-structure nodes, are inclusive. From this we are able to deduce that there is a congruence for the extended language.

Part II is devoted to applications of the congruence. The conditions under which we may re-order the evaluation of arguments to functions are deduced in Chapter 5. We may evaluate the argument x first, in the expres-

sion $f\,x$, if and only if f is completely strict, *i.e.* $f\,\bot = \bot$ and $f\,\underline{\underline{?}} = \underline{\underline{?}}$. This is a stronger condition than that generally supplied by strictness abstract interpretations. An interesting observation is that the language with which we are working is not confluent. The confluence of the λ-calculus is shown to hold by the Church-Rosser Theorem [Barendregt 81, Theorem 11.1.10, page 282]. We are unable to show a similar theorem for our language as there exist two possible reductions of $\underline{\underline{?}} + \bot$; it may be either $\underline{\underline{?}}$ or $\bot$.

In the operational semantics of Chapters 2 and 4, indirection nodes are inserted at the end of each reduction step. This preserves sharing. Other methods have been proposed to avoid the use of indirection nodes, and we investigate these in Chapter 6. In the process of proving the alternative methods equivalent it will be observed that individual reduction steps may fail to terminate. A necessary and sufficient condition for this to occur is that there is a spine cycle, and it is not then possible to determine the root of the next reduction.

In Chapter 7 we show that the G-machine is equivalent to the operational semantics of Chapter 4. Because the operational semantics is congruent to the denotational semantics, we may say that the G-machine is a correct implementation of the language specified by the denotational semantics.

One of the principal areas of optimization in the G-machine is its use of the V stack to hold temporary basic values. In Chapter 8 some of these optimized continuations are shown to be equivalent to the original, unoptimized, continuations. This process involves showing a form of operational equivalence holds; this is related to graph isomorphism. The modified definition of graph isomorphism allows us to relate graphs that differ only on unreachable nodes within the graph or which differ by having indirection nodes elided. Graph isomorphism is extended to allow us to refer to G-machine states and continuations that are graph isomorphic.

A summary of the results derived in this thesis appear in Chapter 9.

Part I

Congruence

Chapter 2

Semantic Models

In this chapter we define the syntax of the language and informally describe its intended semantics. Next a denotational semantics is given. Proofs about the results produced by a program would typically be proved in a denotational setting. An operational semantics for graph reduction is then given. This can be viewed as an implementation of the language. Operational properties of the program, such as space and time complexity, would typically be investigated using the operational semantics. We prove the congruence of the denotational and operational semantic specifications in Chapter 3.

2.1 A Simple Functional Language

In this thesis we shall consider a very simple lazy functional programming language. Its abstract syntax is given in Figures 2.1 and 2.2. A program in this language is then an expression $[\![E]\!]$ and a set of rewrite rules $[\![\Delta]\!]$. Each rewrite rule associates a combinator $[\![\Gamma]\!]$ with an identifier. The combinator $[\![\Gamma]\!]$ is a function of at least one argument and has a body $[\![E]\!]$. The intention is that a free variable of the rewrite rule $[\![I = \Gamma]\!]$ should be the name of another rewrite rule. This means that we may treat each rewrite rule as a combinator as its free variables are in fact constants. It is this important property that allows combinator graph reduction to occur without closures. Another observation is that each definition is a function and not a constant. This is because a combinator reducer would treat constant definitions differently, see [Peyton Jones 87, Section 18.6, page 311]. Since we are interested in the operation of combinator reduction it seems reasonable to simplify our language to a minimum.

Π	$\in$ Prog	(Programs)
Δ	$\in$ Defs	(Function Definitions)
Γ	$\in$ Comb	(Combinator Bodies)
E	$\in$ Exp	(Expression)
B	$\in$ Basic	(Basic Values)
I	$\in$ Ide	(Identifiers)

Figure 2.1: Syntactic Categories

$$
\begin{array}{lcl}
\Pi & ::= & \mathrm{E} \textbf{ where } \Delta \\
\\
\Delta & ::= & \Delta_0 \textbf{ and } \Delta_1 \\
 & | & \mathrm{I} = \Gamma \\
\\
\Gamma & ::= & \lambda \mathrm{I}.\mathrm{E} \\
 & | & \lambda \mathrm{I}.\Gamma \\
\\
\mathrm{E} & ::= & \mathrm{I} \\
 & | & \mathrm{B} \\
 & | & \mathrm{E}_0\,(\mathrm{E}_1)
\end{array}
$$

Figure 2.2: Abstract Syntax

In passing we note there exist transformations that allow the translation of general functional programs into the restricted form we present here. The two well known ones are those of Hughes [1982a, 1982b] and Johnsson [1985], both of which are described in more detail by Peyton Jones [1987]. The aim of both techniques is to reduce a program to the restricted form of combinators, as there exist efficient implementations of such languages.

In Figure 2.1, the names of the syntactic domains and typical elements of these domains are given. In Figure 2.2 the syntax is defined. A program $[\![\Pi]\!]$ is an expression $[\![\mathrm{E}]\!]$ and a set of combinator definitions $[\![\Delta]\!]$. The combinator definitions introduce bindings of the form $[\![\mathrm{I} = \Gamma]\!]$. $[\![\mathrm{I}]\!]$ is an identifier and $[\![\Gamma]\!]$ is the combinator that is bound to $[\![\mathrm{I}]\!]$. Each combinator takes at least one argument because Γ is defined as either $\lambda \mathrm{I}.\Gamma$ or $\lambda \mathrm{I}.\mathrm{E}$. An expression is either an identifier $[\![\mathrm{I}]\!]$; a basic value $[\![\mathrm{B}]\!]$ or an application of two subexpressions $[\![\mathrm{E}_0\,\mathrm{E}_1]\!]$.

It can be seen, in Figure 2.2, there has to be at least one function definition. This may be overcome by amending the syntax with a second pro-

duction for Π, of $\Pi ::= \mathbf{E}$. We have not done this as our proofs in Chapter 3 will become even more complex.

2.2 Notations for Denotational Semantics

The notation used in denotational semantics varies from author to author. In some ways the notation used by Stoy [1977] is cumbersome, and occasionally ambiguous, for the material presented in this thesis. Therefore whilst the majority of the notation comes from [Stoy 77], other parts are derived from functional programming notation.

Syntactic objects are denoted by Greek capitals (E, B, ...) and are elements of domains denoted by short names (Exp, Basic, ...). Corresponding lower case Greek letters (ε, β, ...) denote appropriate semantic values, which come from domains denoted by ($\mathbf{E}$, $\mathbf{B}$, ...); large curly letters ($\mathcal{E}$, $\mathcal{B}$, ...) are used for the "valuation" functions which map syntactic objects to the values they denote.

We shall presume that the domains are defined using complete partial orders. A presentation of this model is available [Schmidt 86, Section 6.5]. The results derived in this paper will of course hold for the alternative definition of the domains as a complete lattice with an element $\top$. Each domain includes an *error element*, denoted by $\underline{\underline{?}}$. As usual $\bot \sqsubseteq \underline{\underline{?}}$, but $\underline{\underline{?}}$ is incomparable with all other elements. If an elemeut is projected into a subdomain where it doesn't belong, it is mapped to $\underline{\underline{?}}$; conversely, the $\underline{\underline{?}}$ element of each subdomain is mapped to $\underline{\underline{?}}$ in the sum. We shall refer to $\bot$ and $\underline{\underline{?}}$ as *improper* elements of a domain; the predicate $Proper(x)$ will be true piovided x is neither $\bot$ nor $\underline{\underline{?}}$.

$\mathbf{D_0}+\mathbf{D_1}$ denotes the *separated* sum of the domains $\mathbf{D_0}$ and $\mathbf{D_1}$. If $\mathbf{D}$ is the separated sum of $\mathbf{D_0}$ and $\mathbf{D_1}$, with $\delta \in \mathbf{D_0}$, then δ in $\mathbf{D}$ is the corresponding element of the sum domain $\mathbf{D}$. If $\delta \in \mathbf{D}$ then $\delta|\mathbf{D_0}$ denotes the following element of $\mathbf{D_0}$:

- if $\delta \equiv \bot_{\mathbf{D}}$ then $(\delta|\mathbf{D_0}) \equiv \bot_{\mathbf{D_0}}$;
- if δ corresponds to an element δ_0 of $\mathbf{D_0}$ then $(\delta|\mathbf{D_0}) \equiv \delta_0$;
- otherwise $(\delta|\mathbf{D_0}) \equiv \underline{\underline{?}}_{\mathbf{D_0}}$.

If $\delta \in \mathbf{D}$ then $\delta \in \mathbf{D_0}$ is the following element of the truth value domain $\mathbf{T}$:

- if $\delta \equiv \bot_{\mathbf{D}}$ then $(\delta \in \mathbf{D_0}) \equiv \bot_{\mathbf{T}}$;

- if δ corresponds to an element δ_0 of $\mathbf{D}_0$ then $(\delta \in \mathbf{D}_0) \equiv \mathit{true}$;
- otherwise $(\delta \in \mathbf{D}_0) \equiv \mathit{false}$.

For $\tau \in \mathbf{T}$ (the domain of truth values) and $\delta_0, \delta_1 \in \mathbf{D}$, the expression

$$\tau \longrightarrow \delta_0, \delta_1 \text{ has value } \begin{cases} \delta_0 & \text{if } \tau \text{ is } \mathit{true} \\ \delta_1 & \text{if } \tau \text{ is } \mathit{false} \\ \bot_{\mathbf{D}} & \text{if } \tau \text{ is } \bot_{\mathbf{T}} \\ \stackrel{?}{=}_{\mathbf{D}} & \text{if } \tau \text{ is } \stackrel{?}{=}_{\mathbf{T}} \end{cases}$$

The major change of notation is in the representation of elements of composite domains. Borrowing from functional programming notation we represent an element of the cartesian product domain $(\mathbf{D}_1 \times \ldots \times \mathbf{D}_n)$ by $(\delta_1, \ldots, \delta_n)$, and we can access the m^{th} component by $\mathit{fst} \circ \mathit{snd}^m$. Similarly we provide a new notation for lists of elements of some domain $\mathbf{D}$. The list domain is represented by $\mathbf{D}^*$ which can be expressed as the solution to the following domain equation[1]:

$$\mathbf{D}^* = (\mathbf{D} \times \mathbf{D}^*) + \{\mathit{nil}\}.$$

Elements of this domain can be represented by either of the two notations used by Turner in SASL [1976]. If $\ell \in \mathbf{D}$ and $\phi \in \mathbf{D}^*$ then we can represent (ℓ, ϕ) in $\mathbf{D}^*$ by $\ell : \phi$. Alternatively, a finite list may be represented by $[\ell_0, \ldots, \ell_n]$, which is equivalent to $(\ell_0 : (\ldots(\ell_n : \mathit{nil})\ldots))$.

As list domains form a significant part of the definition of the operational semantics, we define a selection of useful continuous functions over lists in Figure 2.3.

The other change of notation is to represent the updating of environments by $\rho \oplus \{\mathrm{I} \mapsto \varepsilon\}$ which is defined by

$$\rho \oplus \{\mathrm{I} \mapsto \varepsilon\} = \lambda \mathrm{I}'.((\mathrm{I}' = \mathrm{I}) \longrightarrow \varepsilon, \rho[\![\mathrm{I}']\!]).$$

Notice that $\oplus$ is associative. An arid environment is defined by $\rho_{\text{arid}} = \lambda \mathrm{I}.\bot$.

2.3 Standard Semantics

[1] As solutions to recursive domain equations are unique only up to isomorphism, we could use $\cong$ instead of $=$ to express the equation. This notation is not used in this thesis, as it conflicts with Stoy's notation, and we will make considerable use of $\cong$ in other contexts.

$$
\begin{array}{lcl}
[] +\!\!+ ys & = & ys \\
(x : xs) +\!\!+ ys & = & x : xs +\!\!+ ys \\
 & & \\
(x : xs) \,!\, 0 & = & x \\
(x : xs) \,!\, (n+1) & = & xs \,!\, n \\
 & & \\
\#\,[] & = & 0 \\
\#\,(x : xs) & = & 1 + \#\,xs \\
 & & \\
take\,0\,xs & = & [] \\
take\,(n+1)\,(x : xs) & = & x : take\,n\,xs \\
 & & \\
drop\,0\,xs & = & xs \\
drop\,(n+1)\,(x : xs) & = & drop\,n\,xs \\
 & & \\
map\,f\,[] & = & [] \\
map\,f\,(x : xs) & = & f\,x : map\,f\,xs \\
 & & \\
last\,xs & = & xs \,!\, (\#\,xs - 1)
\end{array}
$$

Figure 2.3: Standard List Operations

The object of a denotational semantics is to map syntactic objects to elements of the value domains we wish to use. Figure 2.4 shows that we have a very simple set of domains. The domain **B** is the domain of basic values, which might represent any simple data types we want. For example, $\mathbf{B} = \mathbf{Z} + \mathbf{T}$ would be suitable if we wanted to use integer and boolean values. The domain **E** is then defined to be the separated sum of **B** and F, where F is the domain of functions from **E** to **E**. We define one more domain, **U**, which maps identifiers to expressions.

As the language is a sugared version of the λ-calculus with base values, the semantic functions of Figure 2.5 are straightforward, and are based on Stoy's work [1982]. In Chapter 4 we consider the addition of primitive operations, such as addition, to this semantic definition.

2.4 An Operational Semantics

$$
\begin{array}{lllll}
\varepsilon & \in & \mathbf{E} & = & \mathbf{B} + \mathbf{F} \quad \text{(Expression Values)} \\
\phi & \in & \mathbf{F} & = & [\mathbf{E} \to \mathbf{E}] \quad \text{(Function Values)} \\
\beta & \in & \mathbf{B} & & \quad \text{(Primitive Values)} \\
\rho & \in & \mathbf{U} & = & [\mathrm{Ide} \to \mathbf{E}] \quad \text{(Environments)}
\end{array}
$$

Figure 2.4: Value Domains for Denotational Semantics

$$
\begin{array}{lcl}
\mathcal{P} : \mathrm{Prog} \to \mathbf{E} & & \\
\mathcal{P}[\![\mathrm{E\ where}\ \Delta]\!] & = & \mathcal{E}[\![\mathrm{E}]\!]\, \mathit{fix}(\mathcal{D}[\![\Delta]\!]) \\
 & & \\
\mathcal{D} : \mathrm{Defs} \to \mathbf{U} \to \mathbf{U} & & \\
\mathcal{D}[\![\Delta_0\ \mathbf{and}\ \Delta_1]\!]\,\rho & = & \mathcal{D}[\![\Delta_0]\!]\,\rho \oplus \mathcal{D}[\![\Delta_1]\!]\,\rho \\
\mathcal{D}[\![\mathrm{I} = \Gamma]\!]\,\rho & = & \{\mathrm{I} \mapsto \mathcal{C}[\![\Gamma]\!]\,\rho\} \\
 & & \\
\mathcal{C} : \mathrm{Comb} \to \mathbf{U} \to \mathbf{E} & & \\
\mathcal{C}[\![\lambda \mathrm{I}.\mathrm{E}]\!]\,\rho & = & \lambda\varepsilon.\mathcal{E}[\![\mathrm{E}]\!]\,(\rho \oplus \{\mathrm{I} \mapsto \varepsilon\})\ \mathbf{in}\ \mathbf{E} \\
\mathcal{C}[\![\lambda \mathrm{I}.\Gamma]\!]\,\rho & = & \lambda\varepsilon.\mathcal{C}[\![\Gamma]\!]\,(\rho \oplus \{\mathrm{I} \mapsto \varepsilon\})\ \mathbf{in}\ \mathbf{E} \\
 & & \\
\mathcal{E} : \mathrm{Exp} \to \mathbf{U} \to \mathbf{E} & & \\
\mathcal{E}[\![\mathrm{I}]\!]\,\rho & = & \rho[\![\mathrm{I}]\!] \\
\mathcal{E}[\![\mathrm{B}]\!]\,\rho & = & \mathcal{B}[\![\mathrm{B}]\!]\ \mathbf{in}\ \mathbf{E} \\
\mathcal{E}[\![\mathrm{E}_0\,(\mathrm{E}_1)]\!]\,\rho & = & (\varepsilon_0 \in \mathbf{F}) \to (\varepsilon_0 \mid \mathbf{F})(\varepsilon_1),\ \stackrel{?}{=} \\
 & & \quad \text{where } \varepsilon_i = \mathcal{E}[\![\mathrm{E}_i]\!]\,\rho \quad \text{for } i = 0, 1
\end{array}
$$

Figure 2.5: Semantic Functions

In this section we define an operational semantics for the language. This operational semantics will also be referred to as an interpreter for the language or an implementation of the language. Because we wish to implement the language described in Section 2.3, we use the same syntactic categories and abstract syntax. The operational semantics is defined so that it models graph reduction. It is this important technique that is the basis for the most efficient implementations of lazy functional languages known. See [Johnsson 83, Johnsson 84] and [Fairburn and Wray 87] for further details.

The domains used to define the operational semantics are given in Figure 2.6. We will use the same letters to represent these domains as we did for the denotational semantics of the previous section, although we are now using them to refer to different domains. Each state in S has two components; a rooted graph (with elements in $\mathbf{G} \times \mathbf{L}$) and a global environment

σ	$\in$	$\mathbf{S}$	$=$	$(\mathbf{G} \times \mathbf{L}) \times \mathbf{D}$	(States)
γ	$\in$	$\mathbf{G}$	$=$	$[\mathbf{L} \to \mathbf{N}]$	(Graphs)
ν	$\in$	$\mathbf{N}$	$=$	$\mathbf{A} + \mathbf{I} + \mathbf{B} + \mathrm{Ide}$	(Nodes)
		$\mathbf{A}$	$=$	$\mathbf{L} \times \mathbf{L}$	(Application Nodes)
		$\mathbf{I}$	$=$	$\mathbf{L}$	(Indirection Nodes)
β	$\in$	$\mathbf{B}$			(Basic Values)
ρ	$\in$	$\mathbf{U}$	$=$	$[\mathrm{Ide} \to \mathbf{L}]$	(Local Environments)
δ	$\in$	$\mathbf{D}$	$=$	$[\mathrm{Ide} \to \mathrm{Comb}]$	(Global Environments)
ℓ	$\in$	$\mathbf{L}$			(Node Labels)
ϕ	$\in$	$\mathbf{L}^*$			(Spines)
κ	$\in$	$\mathbf{K}$	$=$	$[(\mathbf{G} \times \mathbf{L}) \to (\mathbf{G} \times \mathbf{L})]$	(Continuations)

Figure 2.6: Value Domains for Operational Semantics

(with elements in **D**). The graphs we wish to consider have the following properties:

directed This is so that the edges point in one direction only. Such a graph is often referred to as a digraph.

labelled The domain structure we have chosen uses the labels to represent edges.

finite Both the set of vertices and the set of edges are finite.

rooted We need to maintain the distinction of the root node.

A finite labelled digraph is defined in [Harary 69, pages 8–13]. Our rooted graphs are represented by a graph (**G**), from node labels (in **L**) to nodes (in **N**), and a root node from **L**. This represents the root node of our evaluation.

Nodes are represented by *node labels*. These are represented by the flat domain **L**, which is not further defined. We can think of them informally as pointers into the heap. The contents of a node (from domain **N**) may be one of four types, represented by **A**, **I**, **B** and Ide. The domain **A** represents the application nodes, **I** represents the indirection nodes, whilst **B** and Ide have been encountered in Section 2.3.

This notation to represent graphs may not be immediately obvious. It is similar to the way the store is modelled in a typical denotational semantics. An example of a graph is given in Figure 2.7, and we would model this by the graph γ and root ℓ_0 where

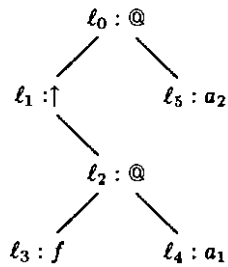

Figure 2.7: Example of a Graph

$$\gamma = \{\ \ell_0 \mapsto (\ell_1, \ell_5)\ \mathbf{in}\ \mathbf{N},\quad \ell_1 \mapsto \ell_2\ \mathbf{in}\ \mathbf{N},\quad \ell_2 \mapsto (\ell_3, \ell_4)\ \mathbf{in}\ \mathbf{N},$$
$$\ell_3 \mapsto f\ \mathbf{in}\ \mathbf{N},\qquad \ell_4 \mapsto a_1\ \mathbf{in}\ \mathbf{N},\quad \ell_5 \mapsto a_2\ \mathbf{in}\ \mathbf{N}\ \}$$

In the pictorial representation @ represents an application node; $\uparrow$ represents an indirection node and f, a_1 and a_2 are identifiers. Each node is labelled $\ell_0 \ldots \ell_5$.

Informally the interpreter performs reductions on a state σ (using the function *Step*) until a state is reached in which no further reductions are possible (tested by *Done*). Of course it is possible that there is no final state and that the interpreter "loops for ever". This is why the *Eval* function of Figure 2.8 is defined in terms of the least fixed point of the sequence of reductions.

Both *Step* and *Done* are defined in terms of a spine which is constructed by the function *Spine*. Informally *Spine* produces a list of pointers to application nodes and a pointer to either a function or a basic value. In the graph drawn in Figure 2.7 the spine is $[\ell_3, \ell_2, \ell_0]$. Notice that the indirection node is not included.

A state σ is terminal when $Done(\sigma)$ holds. This occurs if the first and only element of the spine is a basic value or if the spine represents a function with insufficient arguments to be reduced. Peyton Jones refers to such a state as being in *weak head normal form* (subsequently referred to as WHNF) in his book [Peyton Jones 87, page 198]. The number of arguments required

$$
\begin{array}{lcl}
\mathit{Eval} & = & \mathit{fix}(\lambda\kappa\lambda\sigma.\mathit{Done}(\sigma) \longrightarrow \sigma,\ \kappa(\mathit{Step}(\sigma))) \\
\\
\mathit{Step}((\gamma, \ell_0),\ \delta) & = & ((\gamma' \oplus \{(\phi\,!\,(\mathit{Args}[\![\Gamma]\!] - 1)) \mapsto \ell' \textbf{ in } \mathbf{N}\},\ \ell_0),\ \delta) \\
 & & \quad \text{where } (\ell : \phi) = \mathit{Spine}(\gamma, \ell_0) \\
 & & \quad\quad [\![\Gamma]\!] = \delta((\gamma\,\ell) \mid \mathrm{Ide}) \\
 & & \quad\quad (\gamma', \ell') = C\,[\![\Gamma]\!]\,\rho_{\text{ and }}\phi(\gamma, \ell) \\
\\
\mathit{Done}((\gamma, \ell_0),\ \delta) & = & (\nu \in \mathbf{B} \wedge \#\,\phi = 0) \vee \\
 & & ((\nu \in \mathrm{Ide}) \wedge (\mathit{Args}(\delta\,(\nu \mid \mathrm{Ide})) > \#\,\phi)) \\
 & & \quad \text{where } (\ell : \phi) = \mathit{Spine}(\gamma, \ell_0) \\
 & & \quad\quad \nu = \gamma\,\ell \\
\\
\mathit{Spine}(\gamma, \ell) & = & (\nu \in \mathbf{A}) \longrightarrow \mathit{Spine}(\gamma, \mathit{fst}(\nu \mid \mathbf{A})) +\!\!+ [\ell], \\
 & & (\nu \in \mathbf{I}) \longrightarrow \mathit{Spine}(\gamma, \mathit{Elide}(\gamma, (\nu \mid \mathbf{I}))), \\
 & & (\nu \in \mathbf{B}) \longrightarrow [\ell], \\
 & & (\nu \in \mathrm{Ide}) \longrightarrow [\ell], \\
 & & \quad \text{where } \nu = \gamma\,\ell \stackrel{?}{=} \\
\\
\mathit{Args}[\![\lambda \mathrm{I}.\mathrm{E}]\!] & = & 1 \\
\mathit{Args}[\![\lambda \mathrm{I}.\Gamma]\!] & = & 1 + \mathit{Args}[\![\Gamma]\!]
\end{array}
$$

Figure 2.8: Interpreter

$$
\begin{array}{lcl}
\mathit{New}\,(\gamma) & \text{satisfies} & \gamma(\mathit{New}\,(\gamma)) = \bot \quad \wedge \quad \mathit{New}\,(\lambda\ell.\bot) = \ell_0 \in \mathbf{L} \\
\\
\mathit{Elide}(\gamma, \ell) & = & (\gamma\,\ell \in \mathbf{I}) \longrightarrow \mathit{Elide}(\gamma, (\gamma\,\ell \mid \mathrm{I})), \ell \\
\\
\mathit{Arg}(\gamma, \ell) & = & (\nu \in \mathbf{A}) \longrightarrow \mathit{snd}(\nu \mid \mathbf{A}), \stackrel{?}{=} \\
 & & \quad \text{where } \nu = \gamma\,\ell
\end{array}
$$

Figure 2.9: Auxiliary Definitions

by a function is determined by the auxiliary function *Args*. If a state is non-terminal then a reduction step is performed on that state, using *Step*. A reduction step consists of the following operations.

- Determining the combinator Γ associated with the identifier at the head of the spine.

- Associating the local variables $l_1 \ldots I_n$ of the combinator $\lambda I_1 \ldots \lambda I_n.E$ with the arguments on the spine.
- Constructing an instance of Γ in the heap. This has root ℓ'.
- Overwriting the root of the original redex (ϕ ! $Args\Gamma - 1$) with an indirection to the root (ℓ') of the newly constructed instance of Γ.

The initial state of the interpreter, for program Π, is produced by constructing an initial graph from the expression part of the program, and a global environment from the definition part of the program. This might be regarded as a primitive compilation for the program Π. One could instead use the G-machine as the basis for the operational semantics. We would then have to show that the theorems of Chapter 3 will remain true with this new operational semantic specification. Alternatively, and probably more simply, we can establish an equivalence between the states of the interpreter presented here and those produced by the G-machine. This is done in Chapter 7.

We now describe tbe compilation functions.

$\mathcal{P}$ This function evaluates an initial state whicb is constructed from E and Δ using $\mathcal{E}$ and $\mathcal{D}$.

$\mathcal{D}$ This constructs a global environment in D from Δ. It maps identifiers to combinators, which are represented as syntactic objects.

This is done because we need to determine both the arity of each Γ and the substitution to be performed if the combinator is reduced.

$\mathcal{C}$ Should the combinator Γ be reduced $\mathcal{C}$ performs this reduction. It builds up a local environment ρ in U from the stack ϕ. When it has acquired sufficient arguments it uses $\mathcal{E}$ to construct the graph.

$\mathcal{E}$ This builds a graph to correspond to the expression E. If we have an identifier we must determine whether it is locally or globally defined. The function *dom* is used to test whether I is bound in ρ. If it is we return the original graph and a pointer to the node associated with the local identifier I. Alternatively, in the case of an identifier bound globally, we add a new node to the graph containing the identifier I, and return this new graph and a pointer to the newly created piece of graph.

$$
\begin{array}{lcl}
\mathcal{P} : \text{Prog} \rightarrow \mathbf{S} & & \\
\mathcal{P}\,[\![\text{E where } \Delta]\!] & = & \mathit{Eval}(\mathcal{E}\,[\![\text{E}]\!]\,\rho_{\ \text{arid}}\,\sigma_{\ \text{init}}) \\
 & & \quad \text{where } \sigma_{\ \text{init}} = ((\{\}, \bot), \mathcal{D}\,[\![\Delta]\!]) \\
 & & \\
\mathcal{D} : \text{Defs} \rightarrow \mathbf{D} & & \\
\mathcal{D}\,[\![\Delta_0 \text{ and } \Delta_1]\!] & = & \mathcal{D}\,[\![\Delta_0]\!] \oplus \mathcal{D}\,[\![\Delta_1]\!] \\
\mathcal{D}\,[\![\text{I} = \Gamma]\!] & = & \{\text{I} \mapsto [\![\Gamma]\!]\} \\
 & & \\
\mathcal{C} : \text{Comb} \rightarrow \mathbf{U} \rightarrow \mathbf{L}^* \rightarrow \mathbf{K} & & \\
\mathcal{C}\,[\![\lambda\text{I.E}]\!]\,\rho\,(\ell : \phi) & = & \mathcal{E}\,[\![\text{E}]\!]\,\rho \oplus \{\text{I} \mapsto \ell\} \\
\mathcal{C}\,[\![\lambda\text{I.}\Gamma]\!]\,\rho\,(\ell : \phi) & = & \mathcal{C}\,[\![\Gamma]\!]\,\rho \oplus \{\text{I} \mapsto \ell\}\,\phi \\
 & & \\
\mathcal{E} : \text{Exp} \rightarrow \mathbf{U} \rightarrow \mathbf{K} & & \\
\mathcal{E}\,[\![\text{I}]\!]\,\rho\,(\gamma, \ell) & = & (\text{I} \in \mathit{dom}(\rho)) \longrightarrow (\gamma, \mathit{Arg}(\rho\,[\![\text{I}]\!])), \\
 & & \qquad (\gamma \oplus \{\ell' \mapsto [\![\text{I in N}]\!], \ell') \\
 & & \quad \text{where } \ell' = \mathit{New}(\gamma) \\
\mathcal{E}\,[\![\text{B}]\!]\,\rho\,(\gamma, \ell) & = & (\gamma \oplus \{\ell' \mapsto \mathcal{B}\,[\![\text{B}]\!] \text{ in N}\}, \ell') \\
 & & \quad \text{where } \ell' = \mathit{New}(\gamma) \\
\mathcal{E}\,[\![\text{E}_0\ (\text{E}_1)]\!]\,\rho\,(\gamma_2, \ell_2) & = & (\gamma_0 \oplus \{\ell' \mapsto (\ell_0, \ell_1) \text{ in N}\}, \ell') \\
 & & \quad \text{where } (\gamma_i, \ell_i) = \mathcal{E}\,[\![\text{E}_i]\!]\,\rho\,(\gamma_{i+1}, \ell_{i+1}) \\
 & & \qquad\qquad \text{for } i = 0, 1 \\
 & & \qquad\quad \ell' = \mathit{New}(\gamma_0)
\end{array}
$$

Figure 2.10: Compiling Functions

If the expression is the syntactic representation of a basic value we build a new node in the graph and return both this graph and a pointer to the new node.

Finally, for an application we recursively invoke the semantic function $\mathcal{E}$ on the subexpressions and then construct a new node that represents an application of the two subexpressions. We then return this graph along with a pointer to the application.

Notice that *every* reduction is performed with an indirection node overwriting the root node. This is not necessary for lazy reduction, and one improvement incorporated in Johnsson's G-machine is to overwrite the orig-

inal application node with the contents of tbe new one, whenever the function body is an application. On balance, I feel that the simplicity of the current machine assists the understanding of the correctness proofs. We return to this subject in Chapter 6.

To model shared computations properly we require a store. In the operational semantics this is labelled $\gamma \in G$, and informally it corresponds to the heap in a graph reduction implementation. It is interesting to note that the store we use is less complicated than the general model provided in [Stoy 77, Chapter 12]. We have no need to use *Map* and *Area*, because the store is used in a restricted way by the operational semantics. This implies that we may use a simple version of the function *New*, defined in Figure 2.9. It produces new node labels, in L, that do not clash with any that are present in the current grapb γ. In a real programming environment the function *New* would be a heap allocator of some kind. In this specification we ignore the particular choice of heap allocation algorithm, provided that it can create new node labels for us. The function *Elide* removes indirection nodes which may be used to refer to a node. The function *Arg* is used to select the argument component of an application node.

2.5 Related Work

Denotational semantics for lazy functional languages have been given by Stoy in [1981]. This definition is in some ways too general for our proposed implementation method as it has general definitions within expressions. In other ways it lacks some of the features we wish to explore; such as higher-order built-in functions and structured data objects. Another denotational semantics is provided by Meira for KRC in his thesis [1985]. This incorporates a printer mechanism for structured data objects and higher-order built-in functions.

In his thesis [1987], Augustsson provides a denotational semantics for a lazy functional language which includes used-defined structured data types and pattern-matching. This makes the language definition quite complicated and he therefore omits a congruence proof to establish the correctness of the G-machine.

Formal operational semantics for lazy functional programming languages are easily given for normal order reduction semantics in a term rewriting system. Graph reduction for tbe λ-calculus is explored by Wadsworth in his thesis [1971]. A formal operational semantics for a language similar to

that investigated in this thesis is provided in Johnsson's thesis [1987]. It is a simplified version of Johnsson's operational semantics that we investigate in Chapter 7. An alternative definition, based on a fixed set of combinators, is provided by Turner in [1979a, 1979b].

2.6 Conclusion

We have presented both a denotational and an operational semantic model for a simple lazy functional programming language. The language is sufficiently simple that both semantics are reasonably straightforward. At the same time the language is sufficiently powerful that we would hope to be able to program with it, provided built in functions are included.

We now wish to establish the equivalence or *congruence* of these two definitions. This is the subject of the next chapter.

Chapter 3

Congruence Proof

In this chapter we establish a congruence between the results produced by the denotational and operational semantics. This congruence is not equality since the results of each semantics are from different domains, and these domains are not isomorphic. As we shall see, there is a way to convert final states from the operational semantics to values in the denotational semantics domain. This is not sufficient to make the denotational semantics fully abstract; this is a consequence of Plotkin's counter-example [1977].

In Figure 3.1 we present the proof structure in tabular form, as a guide to the dependencies within the proof. In the text of this chapter each theorem is stated first; any lemmas or subsidiary results are stated and proved; and finally a proof is given.

3.1 The Congruence Proof

We wish to show that the operational semantics of Section 2.4 implements the language defined by the denotational semantics in Section 2.3. Before doing so we introduce the diacritical convention. Because the two specifications under consideration will often use the same names for the same concepts, we distinguish them by accenting them. The more abstract item is normally denoted by an acute accent, whilst the more concrete uses a grave accent. The pair $(\acute{\alpha}, \grave{\alpha})$ is represented by the shorthand $\hat{\alpha}$.

Because the results from the denotational semantics and the interpreter are from different domains, we introduce a derepresentation function E from $\grave{S}$ to $\acute{E}$.

Theorem 3.3 This proves that $\acute{\mathcal{P}}[\![\Pi]\!] = E(\grave{\mathcal{P}}[\![\Pi]\!])$, provided that $\acute{\mathcal{P}}[\![\Pi]\!] \in \mathbf{B}$. It is established by fixpoint induction on the state. It requires

Theorem 3.4 This proves that $\acute{\mathcal{P}}[\![\Pi]\!] \sqsupseteq E(\grave{\mathcal{P}}[\![\Pi]\!])$. It requires

Theorem 3.7 This proves that an individual reduction step on a state will produce a new state that approximates the old one, under derepresentation. It requires

Lemma 3.8 Indirection Node Equivalence Lemma.

Lemma 3.9 Spine Derepresentation Lemma.

Lemma 3.10 Combinator Substitution Lemma.

Theorem 3.12 This proves that $\acute{\mathcal{P}}[\![\Pi]\!] \sqsubseteq E(\grave{\mathcal{P}}[\![\Pi]\!])$, provided $\acute{\mathcal{P}}[\![\Pi]\!] \in \mathbf{B}$. It is established by structural induction on the program, using inclusive predicates. It requires

Lemma 3.2 Proves that derepresentation of a combinator name in the operational semantics is the same value as that supplied by the denotational semantics, for a given syntactic structure Δ.

Theorem 3.17 This proves that the predicates e and f exist. It requires

Lemma 3.18 Partial Predicate Projection Lemma.

Lemma 3.19 Partial Predicate Injection Lemma.

Corollary 3.20 Proves that the predicates establish the required approximation.

Theorem 3.21 Proves that all initial expressions satisfy the predicates. It requires

Lemma 3.22 Proves that the predicates imply the correct behaviour for application nodes.

Figure 3.1: Outline of the Proof Structure

Definition 3.1

$$
\begin{aligned}
E((\gamma, \grave{\iota}), \grave{\delta}) \;=\; & (\grave{\nu} \in \mathrm{Ide}) \longrightarrow acute\,\grave{\delta}\,(\grave{\nu} \mid \mathrm{Ide}),\\
& (\grave{\nu} \in \mathbf{B}) \longrightarrow (\grave{\nu} \mid \mathbf{B})\ \mathbf{in}\ \acute{\mathbf{E}},\\
& (\grave{\nu} \in \grave{\mathbf{I}}) \longrightarrow E((\gamma, \grave{\nu} \mid \grave{\mathbf{I}}), \grave{\delta}),\\
& (\grave{\nu} \in \grave{\mathbf{A}}) \wedge (\acute{\varepsilon}_0 \in \acute{\mathbf{F}}) \longrightarrow (\acute{\varepsilon}_0 \mid \acute{\mathbf{F}})(\acute{\varepsilon}_1),\\
& \qquad\qquad \stackrel{?}{=}\\
& \quad \text{where } \grave{\nu} = \gamma\,\grave{\iota}\\
& \qquad\quad \acute{\varepsilon}_i = E((\gamma, \grave{\iota}_i), \grave{\delta})\\
& \qquad\quad (\grave{\iota}_0, \grave{\iota}_1) = (\grave{\nu} \mid \grave{\mathbf{A}})
\end{aligned}
$$

$$
acute(\delta) \;=\; fix(acute'(\delta))
$$

$$
acute'\,\grave{\delta}\,\acute{\rho}\,[\![\mathrm{I}]\!] \;=\; \acute{C}\,(\grave{\delta}\,[\![\mathrm{I}]\!])\,\acute{\rho}
$$

The auxiliary function *acute* is used to construct the denotational semantics' environment from the interpreter's environment. That this is true is shown by Lemma 3.2.

Lemma 3.2

For all Δ *in* Defs

$$
fix(\acute{\mathcal{D}}\,[\![\Delta]\!]) = acute(\grave{\mathcal{D}}\,[\![\Delta]\!]).
$$

Proof of Lemma 3.2

A simple structural induction, on Δ, establishes that

$$
\acute{\mathcal{D}} = acute' \circ \grave{\mathcal{D}},
$$

from which we have

$$
fix \circ \acute{\mathcal{D}} = fix \circ acute' \circ \grave{\mathcal{D}} = acute \circ \grave{\mathcal{D}}.
$$

□

The function E will produce a value within the domain $\dot{\mathrm{E}}$ for any interpreter state σ in $\grave{\mathrm{S}}$. So we say that σ and σ' have the same meaning if and only if $E(\sigma) = E(\sigma')$. Notice that this is a denotational equivalence, and that different algorithms for computing the same function will be equal under this equivalence. An operational equivalence is defined and used in Chapter 8.

We may now state the congruence condition as Theorem 3.3.

Theorem 3.3

For all Π *in* Prog, *with* $\dot{\mathcal{P}}[\![\Pi]\!] \in \mathrm{B}$,

$$\dot{\mathcal{P}}[\![\Pi]\!] = E(\dot{\mathcal{P}}[\![\Pi]\!]).$$

We will see later in this chapter that the restriction to programs which return base values, is associated with the full abstraction problem [Plotkin 77]. The proof of Theorem 3.3 appears to require two separate proofs, each showing that one definition approximates the other[1]. The reason why such a duplication of effort is necessary is that the two techniques used do not generalize to give the complete congruence of Theorem 3.3. It is instructive to consider where the use of complete, rather than partial congruence causes each method to break down, and this is included in the respective proofs.

3.2 Analysis of the Interpreter

The most obvious method to use to prove Theorem 3.3 is *fixpoint induction* on the interpreter. For this method we use the derepresentation function E, that takes an interpreter state to a value in the denotational semantics' domain. Thus we would hope that during the execution of a program by the interpreter, the derepresentation of the interpreter states would remain constant. From this we would like to conclude that the interpreter always produces the same answer as the denotational semantics. The snag occurs when the interpreter fails to produce an answer, presumably when it is in an "infinite loop". In this case, will the denotational semantics provide the answer $\perp$, or do there exist programs which cause the interpreter to loop forever, but which give proper values in the denotational semantics?

[1] We say "appears" because, in the absence of a meta-proof, we are unable to state categorically that we need two separate proofs.

To avoid these hard questions (at least, until Section 3.3) we shall consider a weaker version of Theorem 3.3.

Theorem 3.4

For all Π *in* Prog

$$\hat{\mathcal{P}}\,[\![\Pi]\!] \sqsupseteq E(\hat{\mathcal{P}}\,[\![\Pi]\!]).$$

That is, for all programs that our syntax of Figure 2.2 defines, the interpreter of Section 2.4 will produce an answer that approximates the semantic values given by the denotatioual semantics of Section 2.3. One way to paraphrase Theorem 3.4 is to say that the interpreter of Section 2.4 is heading in the right direction, because it never produces an answer that disagrees with that of the denotational semantics. Obviously this is a useful result to have about an interpreter, as it allows us to believe any result it may produce.

The method we will use to establish Theorem 3.4 is ***fixpoint induction*** on the interpreter [Manna *et al.* 73]. For fixpoint induction to work we require that the predicate be *inclusive*, see [Stoy 77, page 216].

Definition 3.5

An assertion $q(x)$ is inclusive, if and only if, for all directed X

$$\bigwedge\{q(x), x \in X\} \Rightarrow q(\bigsqcup X).$$

We recall from [Manna *et al.* 73] that, for any inclusive predicate q and monotone function H, with:

1. $q(\bot)$ and
2. $q(\kappa) \Rightarrow q(H(\kappa))$,

it is the case that $q(\mathit{fix}(H))$ also holds.

After looking at the definition of the interpreter, we define q and H.

Definition 3.6

$$\begin{array}{lcl} q(\kappa) & = & E \circ \kappa \sqsubseteq E \\ H & = & \lambda\kappa\lambda\sigma.\mathit{Done}(\sigma) \longrightarrow \sigma, \kappa(\mathit{Step}(\sigma)) \end{array}$$

With these definitions in mind we now seek to prove that Theorem 3.4 holds. In order to establish this we will eventually need to show that $E(\sigma) \sqsupseteq E(\mathit{Step}(\sigma))$ whenever σ is non-terminal. In fact we can show the equivalence of these values. That is: performing a single reduction step on a non-terminal state will not change the meaning of a graph.

Theorem 3.7

For all σ in $\grave{\mathrm{S}}$, such that $\mathit{Done}(\sigma)$ does not hold:

$$E(\sigma) \sqsupseteq E(\mathit{Step}(\sigma)).$$

A reduction step, implemented by the function *Step*, consists of two parts. In the first part we construct a graph to represent the body of the function we are reducing. The second part consists of overwriting the original root of the reduction with an indirection node to the root of the new piece of graph. This means that the proof of Theorem 3.7 can be split into two parts, the first shows that the use of indirection nodes does not change the meaning of the graph, and the second, shows that the use of $\grave{\mathcal{E}}$ makes a piece of graph with the same meaning as the original graph.

Lemma 3.8

If $E((\gamma, \ell), \delta) = E((\gamma, \ell'), \delta)$ then:

$$E((\gamma \oplus \{\ell \mapsto \ell' \text{ in } \grave{\mathrm{N}}\}, \ell), \delta) \sqsubseteq E((\gamma, \ell), \delta).$$

This says that given two nodes in a graph with the same denotational meaning under E, we may replace one of them by an indirection node to the other. Lemma 3.8 is a general result about indirection nodes under such circumstances, and includes the case where a node is overwritten with a pointer to itself. The new meaning of the overwritten node is clearly $\perp$, although this may not have been the original meaning of the node.

Proof of Lemma 3.8

Let $\gamma' = \gamma \oplus \{\ell \mapsto \ell' \text{ in } \grave{\mathrm{N}}\}$. If we redefine E in terms of an explicit fixpoint, so that $E = \mathit{fix}(E')$, we have:

$$
\begin{array}{rcl}
E'\,e((\grave{\gamma}, \grave{\ell}), \grave{\delta}) = & (\grave{\nu} \in \mathrm{Ide}) & \longrightarrow \ \mathit{acute}\,\grave{\delta}\,(\grave{\nu} \mid \mathrm{Ide}), \\
& (\grave{\nu} \in \mathbf{B}) & \longrightarrow \ (\grave{\nu} \mid \mathbf{B})\ \mathbf{in}\ \acute{\mathbf{E}}, \\
& (\grave{\nu} \in \grave{\mathbf{I}}) & \longrightarrow \ e\,((\grave{\gamma}, \grave{\nu} \mid \grave{\mathbf{I}}), \grave{\delta}), \\
& (\grave{\nu} \in \grave{\mathbf{A}}) & \longrightarrow \ (\acute{\epsilon}_0 \in \acute{\mathbf{F}}) \longrightarrow \\
& & \qquad (\acute{\epsilon}_0 \mid \acute{\mathbf{F}})\,\acute{\epsilon}_1, \underline{\overset{?}{=}}, \\
& & \qquad \underline{\overset{?}{=}} \\
\text{where} & \grave{\nu} & = \ \gamma\,\grave{\ell} \\
& \acute{\epsilon}_i & = \ e\,((\grave{\gamma}, \grave{\ell}_i), \grave{\delta}) \\
& (\grave{\ell}_0, \grave{\ell}_1) & = \ (\grave{\nu} \mid \grave{\mathbf{A}}).
\end{array}
$$

We may now show that the lemma holds by fixpoint induction.

Let q be defined as:

$$q(e) \Leftrightarrow \forall r\,.\,e((\gamma', r), \delta) \sqsubseteq E((\gamma, r), \delta).$$

As the base case is trivially satisfied we consider the inductive step. Let $\varepsilon = e((\gamma', r), \delta)$ and $\varepsilon' = E'\,e((\gamma', r), \delta)$. Assume inductively that $e((\gamma', \ell), \delta) \sqsubseteq E((\gamma', \ell), \delta)$ for all $\ell \in \mathbf{L}$. Consider the cases of $\nu = \gamma'\,r$:

1. $(\nu \in \mathrm{Ide})$
   then $\varepsilon = \mathit{acute}\,\delta\,(\nu \mid \mathrm{Ide}) = \varepsilon'$.
2. $(\nu \in \mathbf{B})$
   then $\varepsilon = (\nu \mid \mathbf{B})\ \mathbf{in}\ \acute{\mathbf{E}} = \varepsilon'$.
3. $(\nu \in \grave{\mathbf{I}})$
   We now consider two cases for r:
   (a) $(r = \ell)$ Then $\gamma'\,r = \ell'$ in $\grave{\mathbf{N}}$. But
   $$\varepsilon' = e((\gamma', \ell'), \delta) \sqsubseteq E((\gamma', \ell'), \delta),$$
   by our inductive hypothesis. But $\varepsilon' \sqsubseteq E((\gamma, \ell), \delta)$ by hypothesis of lemma.
   (b) $(r \neq \ell)$ In this case $\varepsilon' = e((\gamma', \nu \mid \grave{\mathbf{I}}), \delta)$, which satisfies the inductive hypothesis.
4. $(\nu \in \grave{\mathbf{A}})$
   This is the same as case 3, only we need not consider the case where $r = \ell$.

□

The reason that we have not proved equality in Lemma 3.8 is that when $\ell = \ell'$ we create a cycle where none existed before. This results in the left hand side of the lemma statement being $\perp$, while the right hand side remains unchanged. Of course, all of our indirection nodes are inserted so that this can not happen, but the statement of our revised lemma would have to include restrictions.

Before proving our assertion that rewriting a function body preserves the meaning of a graph, we first establish a derepresentation lemma for a state in terms of its spine.

Lemma 3.9

Suppose that $[\ell'_0, \ldots, \ell'_m] = Spine(\gamma, \ell'_m)$ *and that* $\ell_i = snd(\gamma \ell'_i \mid \mathrm{A})$ *for* $1 \leq i \leq m$. *Then*

$$E((\gamma, \ell'_m), \delta) = (E((\gamma, \ell'_0), \delta))\ (E((\gamma, \ell_1), \delta)) \ldots (E((\gamma, \ell_m), \delta)).$$

Proof of Lemma 3.9

By induction on the length of the spine, m.

□

We now state and prove the graph rewriting lemma.

Lemma 3.10

Suppose that $[\ell'_0, \ldots, \ell'_m] = Spine(\gamma, \ell'_m)$ *and that* $\ell_i = snd(\gamma\, \ell'_i \mid \mathrm{A})$ *for* $1 \leq i \leq m$. *Suppose that* $\delta(\gamma\, \ell'_0 \mid \mathrm{Ide}) = [\![\Gamma]\!]$ *and* $\Gamma = \lambda \mathrm{I}_1 \ldots \lambda \mathrm{I}_n.\mathrm{E}$ *with* $m \geq n$. *Then*

$$E((\gamma, \ell'_n), \delta) = E((\hat{C}\, [\![\Gamma]\!]\, \rho_{\mathrm{arid}}\, [\ell'_1, \ldots, \ell'_m](\gamma, \ell'_n)), \delta).$$

Proof of Lemma 3.10

We first observe from the spine derepresentation lemma, Lemma 3.9, that the left hand side becomes

$$(acute(\delta)(\gamma \ell_0' \mid \mathrm{Ide}))\,(E((\gamma, \ell_1), \delta)) \ldots (E((\gamma, \ell_n), \delta)).$$

But $acute(\delta)(\gamma \ell_0' \mid \mathrm{Ide}) = \acute{C}\,[\![\Gamma]\!]\,(acute(\delta))$, which makes the left hand side $\acute{\mathcal{E}}\,[\![\mathrm{E}]\!]\,(acute(\delta) \oplus \acute{\rho})$ where

$$\acute{\rho} = \{\mathrm{I}_1 \mapsto E((\gamma, \ell_1)\delta), \ldots, \mathrm{I}_n \mapsto E((\gamma, \ell_n)\delta)\}.$$

The right hand side becomes $E(\grave{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\grave{\rho}\,(\gamma, \ell_n'), \delta)$, where $\grave{\rho} = \{\mathrm{I}_1 \mapsto \ell_1', \ldots, \mathrm{I}_n \mapsto \ell_n'\}$. A structural induction, on E, now suffices to show the equivalence.

1. $\mathrm{E} = [\![\mathrm{I}]\!] \wedge \mathrm{I} \in dom(\grave{\rho})$
   Without loss of generality, $\mathrm{I} = \mathrm{I}_i$, then by substitution, both sides become $E((\gamma, \ell_i), \delta)$.
2. $\mathrm{E} = [\![\mathrm{I}]\!] \wedge \mathrm{I} \notin dom(\grave{\rho})$
   The left hand side becomes $acute\,\delta\,[\![\mathrm{I}]\!]$, whilst the right hand side is $E((\gamma \oplus \{\ell' \mapsto \mathrm{I}\ \mathbf{in}\ \grave{\mathrm{N}}\}, \ell'), \delta)$, where $\ell' = New(\gamma)$, which is $acute\,\delta\,[\![\mathrm{I}]\!]$.
3. $\mathrm{E} = [\![\mathrm{B}]\!]$
   The left hand side becomes $\mathcal{B}\,[\![\mathrm{B}]\!]\ \mathbf{in}\ \acute{\mathrm{E}}$, whilst the right hand side is $E((\gamma \oplus \{\ell' \mapsto \mathcal{B}\,[\![\mathrm{B}]\!]\ \mathbf{in}\ \grave{\mathrm{N}}\}, \ell'), \delta)$, where $\ell' = New(\gamma)$, which is $\mathcal{B}\,[\![\mathrm{B}]\!]\ \mathbf{in}\ \acute{\mathrm{E}}$.
4. $\mathrm{E} = [\![\mathrm{E}_0\,(\mathrm{E}_1)]\!]$
   Suppose inductively that the Lemma is true for E_i for $i = 0, 1$. *i.e.* for all γ and all r, and for $i = 0, 1$:

$$\begin{aligned} \varepsilon_i &= \acute{\mathcal{E}}\,[\![\mathrm{E}_i]\!]\,(acute(\delta) \oplus \acute{\rho}) \\ &= E(\grave{\mathcal{E}}\,[\![\mathrm{E}_i]\!]\,\grave{\rho}\,(\gamma, r), \delta). \end{aligned}$$

   Then the left hand side becomes $(\varepsilon_0\ \mathbf{e}\ \acute{\mathrm{F}}) \longrightarrow (\varepsilon_0 \mid \acute{\mathrm{F}})\,\varepsilon_1, \underline{?}$. The right hand side is

$$\begin{aligned} &E((\gamma', r), \delta) \\ &\quad \mathbf{where}\ (\gamma_0, r_0) = \grave{\mathcal{E}}\,[\![\mathrm{E}_0]\!]\,\grave{\rho}\,(\gamma_1, \ell) \\ &\qquad\qquad (\gamma_1, r_1) = \grave{\mathcal{E}}\,[\![\mathrm{E}_1]\!]\,\grave{\rho}\,(\gamma, \ell) \\ &\qquad\qquad r = New(\gamma_1) \\ &\qquad\qquad \gamma' = \gamma_0 \oplus \{r \mapsto (r_0, r_1)\ \mathbf{in}\ \grave{\mathrm{N}}.\} \end{aligned}$$

   But $\varepsilon_i = E((\gamma', r_i), \delta)$, for $i = 0, 1$, because of the properties of *New*. Hence we have the result.

□

We have now shown how to relate the derepresented values of indirection nodes by Lemma 3.8 and the relationship between the graph before and after the construction of a combinator body. We now combine these results to prove Theorem 3.7, which states that for any non-terminal state σ:

$$E(\sigma) \sqsupseteq E(\mathit{Step}(\sigma)).$$

Proof of Theorem 3.7

Suppose $\sigma = ((\gamma, \ell'_m), \delta)$. Then, to satisfy $\mathit{Done}(\sigma) = \mathit{false}$, we must have:

1. $[\ell'_0, \ldots, \ell'_m] = \mathit{Spine}(\gamma, \ell'_m)$.
2. $\ell_i = \mathit{snd}(\gamma\, \ell'_i \mid \mathbf{A})$, for $1 \leq i \leq m$.
3. $[\![\Gamma]\!] = \delta\,(\gamma\, \ell'_0 \mid \mathrm{Ide})$.
4. $\mathit{Args}[\![\Gamma]\!] = n$, with $n \leq m$.

We recall that $E((\gamma, \ell'_m), \delta)$ can be expressed as

$$(E((\gamma, \ell'_0), \delta))\ (E((\gamma, \ell_1), \delta)) \ldots (E((\gamma, \ell_m), \delta)).$$

Let $(\gamma', \ell') = \hat{\mathcal{C}}\,[\![\Gamma]\!]\,\rho_{\mathrm{arid}}\,[\ell'_1, \ldots, \ell'_m]\,(\gamma, \ell'_m)$, then, by Lemma 3.10,

$$E((\gamma, \ell'_n), \delta) = E((\gamma', \ell'_n), \delta) = E((\gamma', \ell'), \delta).$$

So by Lemma 3.8 we may deduce

$$E((\gamma, \ell'_n), \delta) \sqsupseteq E((\gamma' \oplus \{\ell'_n \mapsto \ell' \text{ in } \mathbf{N}\}, \ell'_n), \delta).$$

But $E(\sigma) = (E((\gamma, \ell'_n), \delta))(E((\gamma, \ell_{n+1}), \delta)) \ldots (E((\gamma, \ell_m), \delta))$, and

$$E(\mathit{Step}(\sigma)) = (E((\gamma'', \ell'_n), \delta))(E((\gamma'', \ell_{n+1}), \delta)) \ldots (E((\gamma'', \ell_m), \delta)),$$

where $\gamma'' = \gamma' \oplus \{\ell'_n \mapsto \ell' \text{ in } \mathbf{N}\}$. So provided $E((\gamma, \ell'_n), \delta)$ is monotonic, we have $E(\sigma) \sqsupseteq E(\mathit{Step}(\sigma))$. But $E((\gamma, \ell'_n), \delta)$ is monotonic because it has been derepresented to have a value in $\mathbf{E}$, which includes only continuous functions.

□

Finally we are in a position to prove, by fixpoint induction, that the interpreter approximates the denotational value for a program, which is the main result of this section. This theorem is of general use for all interpreters defined using *Done* and *Step*. If, for some E, $E(Step(\sigma)) \sqsubseteq E(\sigma)$ then the operational semantics will satisfy a partial congruence.

Recall that we defined q and H in Definition 3.6 as

$$\begin{array}{lcl} q(\kappa) & = & E \circ \kappa \sqsubseteq E \\ H & = & \lambda\kappa\lambda\sigma.Done(\sigma) \longrightarrow \sigma, \kappa(Step(\sigma)) \end{array}$$

and that Theorem 3.4 states that for all Π in Prog,

$$\dot{\mathcal{P}}[\![\Pi]\!] \sqsupseteq E(\dot{\mathcal{P}}[\![\Pi]\!]).$$

Proof of Theorem 3.4

There are two parts. We first establish that $q(\mathit{fix}(H))$ holds.

First we verify the base case and then we must prove the inductive step.

1. Because $E(\bot(\sigma)) = \bot$ for all σ, $q(\bot)$ holds trivially. Notice that it is at this point that the proof would break down if we attempted to establish total congruence by this method. We would need to show that $E(\bot(\sigma)) = E(\sigma)$ for all σ, which does not define a very useful programming language.
2. Consider $q(H(\kappa))$. This holds if and only if $E(H\,\kappa\,\sigma) \sqsubseteq E(\sigma)$, which, expanding H, is $E(Done(\sigma) \longrightarrow \sigma, \kappa(Step(\sigma))) \sqsubseteq E(\sigma)$. We must consider the alternative values that $Done(\sigma)$ may have.
   (a) $Done(\sigma) = \bot$ or $Done(\sigma) = \underline{?}$.
   Both these cases trivially hold.
   (b) $Done(\sigma)$ is *true*.
   Substituting for $Done(\sigma)$, we have $H\,\kappa\,\sigma = \sigma$, and so
   $$E(H\,\kappa\,\sigma) = E(\sigma).$$
   (c) $Done(\sigma)$ is *false*.
   Substituting for $Done(\sigma)$, we have that
   $$E(H\,\kappa\,\sigma) = E(\kappa(Step(\sigma))).$$
   But, $E(\kappa(Step(\sigma))) \sqsubseteq E(Step(\sigma))$, by the inductive hypothesis $q(\kappa)$. Furthermore, $E(Step(\sigma)) \sqsubseteq E(\sigma)$, from Theorem 3.7.

Hence we may conclude $q(fix(H))$. This can be restated as

For all σ in $\grave{\mathrm{S}}$ $E(Eval(\sigma)) \sqsubseteq E(\sigma)$.

So, to prove Theorem 3.4, we must show that the initial state

$$\grave{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\grave{\rho}_{\,\mathrm{arid}}\,\grave{\sigma}_{\,\mathrm{init}},$$

satisfies the approximation condition

$$\acute{\mathcal{E}}\,[\![\mathrm{E}]\!]\,fix(\acute{\mathcal{D}}\,[\![\Delta]\!]) \sqsupseteq E(\grave{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\grave{\rho}_{\,\mathrm{arid}}\,\grave{\sigma}_{\,\mathrm{init}}).$$

By Lemma 3.2, we may relate the two environments, as they are both created from Δ. A consequence of this is that the derepresentation function E now maps the initial state to the denotational value required. This is a corollary of the structural induction of Lemma 3.10, when both ρ and $\grave{\rho}$ are arid.

□

At this point we know that $\acute{\mathcal{P}}\,[\![\Pi]\!] \sqsupseteq E(\grave{\mathcal{P}}\,[\![\Pi]\!])$, although we could have proved something slightly stronger, namely:

Proposition 3.11

For all Π *in* Prog, *if* $E(\grave{\mathcal{P}}\,[\![\Pi]\!]) = \epsilon$ *and* $\epsilon \neq \bot$ *then*

$$\acute{\mathcal{P}}\,[\![\Pi]\!] = \epsilon.$$

In [Schmidt 86, Section 10.7] this property is referred to as the *faithfulness* of the operational semantics with respect to the denotational semantics. However, this is still not strong enough to show complete congruence. The reason becomes clear if we define $\acute{\mathcal{P}}\,[\![\Pi]\!] = \bot$. Then every program is given the semantic value $\bot$ by the interpreter, which certainly satisfies Proposition 3.11, but does not satisfy the complete congruence of Theorem 3.3. A more subtle problem occurs when we consider whether an applicative order operational semantics would correctly implement our denotational semantics. It clearly does not, although it does satisfy Proposition 3.11. It is only when we can show that our interpreter does not produce $\bot$ when the denotational semantics give a proper value, that we can feel confident about the implementation of the interpreter. This is the object of the next section.

3.3 Predicates for a Structural Induction

We now wish to prove that the denotational semantics approximates the result provided by the interpreter. We will see later that it is only possible to compare base values, and so the formal statement of Theorem 3.12 is restricted to this case.

Theorem 3.12

For all Π *in* Prog, *with* $\hat{\mathcal{P}}[\![\Pi]\!] \in \mathbf{B}$,

$$\hat{\mathcal{P}}[\![\Pi]\!] \sqsubseteq E(\hat{\mathcal{P}}[\![\Pi]\!]).$$

The inductive principle used in this section is that of domain induction. In this way the denotational semantics progressively gives better and better approximations to the correct value. Informally the first approximation to the domain $\mathbf{E}$ is $\mathbf{E}_0 = \bot + \mathbf{B}$. We therefore represent all functions by $\lambda x.\bot$, although the basic values in $\mathbf{B}$ continue to be mapped to the correct element in $\mathbf{B}$. The next approximation is $\mathbf{E}_1 = [\mathbf{E}_0 \to \mathbf{E}_0] + \mathbf{B}$. With this domain we are able to represent functions from $\mathbf{B}$ to $\mathbf{B}$ accurately, but higher order functions are still only represented by an approximation.

There are two common models of domains. The first is Scott's explicit construction [1973], which will result in something like the $\mathbf{D}_\infty$ model. In this case we must make explicit the projections and injections between $\mathbf{E}_i$ and $\mathbf{E}_{i+1}$, so that $\mathbf{E}_i$ is correctly embedded in $\mathbf{E}_{i+1}$. The alternative is the $\mathcal{P}\omega$ model where this detail has been taken care of by using retracts. The $\mathcal{P}\omega$ model [Scott 76] has a lattice theoretic domain structure. It therefore also has a $\top$ element. Barendregt [1981] shows how to derive a model based on algebraic complete partial orders instead of complete lattices. As Barendregt's model has less clutter we use his model. For a comparison of the $\mathbf{D}_\infty$ and $\mathcal{P}\omega$ models when $\mathbf{D} = [\mathbf{D} \to \mathbf{D}]$ see [Wadsworth 76].

The technique used is that of Milne [1974], although another equivalent solution is provided by Gordon [1973]. The method developed by Milne involves the definition of inclusive predicates, that specify a relation between values produced by each definition. It is usual to have one predicate for each domain in the denotational definition, but we dispense with the one for the domain $\mathbf{B}$ because $\mathbf{B}$ occurs as a result for the interpreter too. They state that the approximation condition holds for particular classes of objects produced by the interpreter.

The first thing to do is define a predicate e on $\hat{\varepsilon}$. The intention is that $e(\hat{\varepsilon})$ will hold whenever $\hat{\varepsilon} \sqsubseteq E(\hat{\varepsilon})$ and $\varepsilon \in \mathbf{B}$. This is established by Corollary 3.20.

Definition 3.13

$$
\begin{array}{lcll}
e(\hat{\varepsilon}) & \Leftrightarrow & (\hat{\varepsilon} \equiv \bot) & \longrightarrow \ \hat{\varepsilon} \equiv \bot, \\
 & & (\hat{\varepsilon} \equiv \underline{?}) & \longrightarrow \ \hat{\varepsilon} \sqsubseteq \underline{?}, \\
 & & (\mathit{IsBasic}(\hat{\varepsilon})) & \longrightarrow \ (\hat{\varepsilon} \in \mathbf{B}) \wedge \hat{\varepsilon} \sqsubseteq E(\hat{\varepsilon}), \\
 & & (\mathit{IsFunction}(\hat{\varepsilon})) & \longrightarrow \ (\hat{\varepsilon} \in \mathbf{F}) \wedge f((\hat{\varepsilon} \mid \mathbf{F}), \hat{\varepsilon}), \\
 & & \mathit{false} &
\end{array}
$$

$$
f(\hat{\varepsilon}) \ \Leftrightarrow \ \bigwedge\{e(\hat{\varepsilon}(\hat{\alpha}), \mathit{Eval}(\hat{\varepsilon}_\alpha)) \mid \mathit{Applied}(\hat{\varepsilon}_\alpha, \hat{\varepsilon}, \hat{\alpha}) \wedge e(\hat{\alpha}, \mathit{Eval}(\hat{\alpha}))\}
$$

$$
\mathit{IsBasic}((\gamma, \ell), \delta) = (\mathit{Spine}(\gamma, \ell) = [\ell]) \wedge (\gamma\, \ell \in \mathbf{B})
$$

$$
\mathit{IsFunction}((\gamma, \ell), \delta) = (\mathit{Spine}(\gamma, \ell) = (\ell' : \phi)) \wedge (\gamma\, \ell' \in \mathrm{Ide})
$$

$$
\begin{array}{l}
\mathit{Applied}((\gamma_{\phi(\alpha)}, \ell_{\phi(\alpha)}), \delta)((\gamma_\phi, \ell_\phi), \delta)((\gamma_\alpha, \ell_\alpha), \delta) = \\
\quad (\forall \ell \in \mathbf{L} \ . \ ((\nu_\phi = \underline{?}) \wedge \nu_{\phi(\alpha)} = \nu_\alpha) \vee \\
\qquad ((\nu_\alpha = \underline{?}) \wedge \nu_{\phi(\alpha)} = \nu_\phi) \vee \\
\qquad ((\nu_\phi = \nu_\alpha) \wedge \nu_{\phi(\alpha)} = \nu_\phi)) \wedge \\
\quad (\gamma_{\phi(\alpha)}(\mathit{Elide}(\ell_{\phi(\alpha)})) = (\ell_\phi, \ell_\alpha)) \\
\quad \text{where } \nu_\xi = \gamma_\xi\, \ell_\xi \text{ for } \xi = \phi(\alpha), \phi, \alpha
\end{array}
$$

The comparison of non-functional results is straightforward. For comparing two functional results there are two alternative strategies. The first is to define a predicate to compare two environments, one in the denotational semantics and one in the interpreter. This is the solution adopted by Stoy in [1977]. If we were required to prove correct an SECD machine or Curien's CAM [1986], we would probably adopt this approach, because the interpreter environment then corresponds quite closely to that of the denotational semantics. We note that as we would still be dealing with a reflexive domain, we would still need to use an inclusive predicate method. The alternative, used by Stoy in [1981], is to define an auxiliary predicate that expresses the approximation condition between functional values in terms of their value

when applied to arguments that satisfy the approximation condition. This is a natural way to express the approximation condition for a graph reduction interpreter.

We must prove the existence of e and f, since they are not necessarily well defined. This is done by construction. First define a sequence of approximations to e and f, using retractions to limit the domains over which the approximate predicates are defined.

We recall from [Stoy 77, Chapter 7] that the solution to a domain equation can be found by using retractions on the $\mathcal{P}\omega$ model. If this is done for the equation $\mathbf{E} = [\mathbf{E} \to \mathbf{E}] + \mathbf{B}$ we obtain the following retraction

$$\mathsf{E} = \mathit{fix}\,(\lambda \mathsf{E}.(\mathsf{E} \circ\!\!\to \mathsf{E}) \oplus \mathsf{B}),$$

and the domain $\mathbf{E}$ is then the range or retract of the function E on $\mathcal{P}\omega$.

Instead of working directly with elements of $\mathcal{P}\omega$, we may use the alternative definitions of the operators $\otimes$, $\oplus$ and $\circ\!\!\to$, given in [Stoy 77, Chapter 7] as Theorems 7.44, 7.45 and 7.46 respectively. We note that $\oplus$ is being used here to represent a different function from that described in Chapter 2. These results are now reproduced.

Definition 3.14

Suppose that E and F are retractions on the domains $\mathbf{E}$ and $\mathbf{F}$ respectively. Then the following equivalences are observed:

$$\begin{array}{lcll}
(\mathsf{E} \otimes \mathsf{F})\pi & = & (\pi = (\varepsilon, \phi)) & \longrightarrow \ \mathsf{E}(\varepsilon) \times \mathsf{F}(\phi),\ \pi \\
(\mathsf{E} \oplus \mathsf{F})\sigma & = & (\sigma \in \mathbf{E}) & \longrightarrow \ \mathsf{E}(\sigma \mid \mathbf{E})\ \mathbf{in}\ \mathbf{E} + \mathbf{F}, \\
 & & (\sigma \in \mathbf{F}) & \longrightarrow \ \mathsf{F}(\sigma \mid \mathbf{F})\ \mathbf{in}\ \mathbf{E} + \mathbf{F}, \sigma \\
(\mathsf{E} \circ\!\!\to \mathsf{F})\phi & = & \lambda\varepsilon.\mathsf{F}(\phi(\mathsf{E}(\varepsilon))) &
\end{array}$$

Let us define

$$\begin{array}{lcl}
\mathsf{E}_n & = & \mathsf{F}_n \oplus \mathsf{B} \\
\mathsf{F}_0 & = & \bot \\
\mathsf{F}_{n+1} & = & \mathsf{E}_n \circ\!\!\to \mathsf{E}_n
\end{array}$$

Then we note that $(\lambda \mathsf{E}.(\mathsf{E} \circ\!\!\to \mathsf{E}) \oplus \mathsf{B})^n = \mathsf{E}_n$ and so $\mathsf{E} = \bigsqcup_{n=0}^{\infty} \mathsf{E}_n$ *i.e.*, for all $\varepsilon \in \mathbf{E}$

$$\bigsqcup_{n=0}^{\infty} \mathsf{E}_n(\varepsilon) = \varepsilon$$

The sequences of approximate predicates, e_n and f_n, are defined on the retractions of $\mathbf{E}$ and $\mathbf{F}$ specified by E_n and F_n respectively. If $\hat{\varepsilon}$ satisfies e_n, then it is intended that the projection of $\hat{\varepsilon}$ into $\mathbf{E}_n$ satisfies $e(\mathsf{E}_n(\hat{\varepsilon}), \hat{e})$.

Definition 3.15

$$
\begin{array}{lcll}
e_n(\hat{\varepsilon}) & \Leftrightarrow & (\hat{e} \equiv \bot) & \longrightarrow \ \varepsilon \equiv \bot, \\
 & & (\hat{e} \equiv \underline{?}) & \longrightarrow \ \varepsilon \sqsubseteq \underline{?}, \\
 & & (\mathit{IsBasic}(\hat{e})) & \longrightarrow \ (\varepsilon \mathbin{\mathbf{E}} \mathbf{B}) \wedge \varepsilon \sqsubseteq E(\hat{e}), \\
 & & (\mathit{IsFunction}(\hat{e})) & \longrightarrow \ (\varepsilon \mathbin{\mathbf{E}} \mathbf{F}) \wedge f_n(\mathsf{F}_n(\varepsilon), \hat{e}), \\
 & & \mathit{false} & \\
 & & & \\
f_0(\hat{\varepsilon}) & \Leftrightarrow & \mathit{true} & \\
f_{n+1}(\hat{\varepsilon}) & \Leftrightarrow & \bigwedge\{e_n((\mathsf{F}_{n+1}(\varepsilon))(\mathsf{E}_n(\hat{\alpha})), \mathit{Eval}(\hat{e}_\alpha)) \mid & \\
 & & \qquad \mathit{Applied}(\hat{e}_\alpha, \hat{e}, \hat{\alpha}) & \\
 & & \qquad\quad \wedge\, e_n(\mathsf{E}_n(\hat{\alpha}),\ \mathit{Eval}(\hat{\alpha}))\} &
\end{array}
$$

These predicates, in contrast to the definitions of e and f, are clearly well defined as there is no recursion involved in their definition.

We now define Φ. If $\Phi(P, Q)$ holds, then P and Q satisfy the definitions of e and f respectively. And in Theorem 3.17 we will be showing that there are equivalent alternative definitions of e and f.

Definition 3.16

$$
\begin{array}{lcll}
\Phi(P, Q) & \Leftrightarrow & \forall \hat{\varepsilon}\,.\, P(\hat{\varepsilon}) \Leftrightarrow & \\
 & & (\hat{e} \equiv \bot) & \longrightarrow \ \varepsilon \equiv \bot, \\
 & & (\hat{e} \equiv \underline{?}) & \longrightarrow \ \varepsilon \sqsubseteq \underline{?}, \\
 & & (\mathit{IsBasic}(\hat{e})) & \longrightarrow \ (\varepsilon \mathbin{\mathbf{E}} \mathbf{B}) \wedge \varepsilon \sqsubseteq E(\hat{e}), \\
 & & (\mathit{IsFunction}(\hat{e})) & \longrightarrow \ (\varepsilon \mathbin{\mathbf{E}} \mathbf{F}) \wedge Q((\varepsilon \mid \mathbf{F}), \hat{e}), \\
 & & \mathit{false} & \\
 & \wedge & \forall \hat{\phi}\,.\, Q(\hat{\phi}) \Leftrightarrow & \\
 & & \bigwedge\{P(\varepsilon(\hat{\alpha}), \mathit{Eval}(\hat{e}_\alpha)) \mid & \\
 & & \quad \mathit{Applied}(\hat{e}_\alpha, \hat{e}, \hat{\alpha}) & \\
 & & \quad\ \wedge\, P(\hat{\alpha}, \mathit{Eval}(\hat{\alpha}))\} &
\end{array}
$$

We now claim that

$$e(\hat{\varepsilon}) = \bigwedge_{n=0}^{\infty} e_n(\mathsf{E}_n(\varepsilon), \hat{e}) \text{ and } f(\hat{\phi}) = \bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\phi), \hat{\phi})$$

satisfy the definitions of e and f given in Definition 3.13, and hence that these predicates exist by construction. This is stated formally as Theorem 3.17.

Theorem 3.17

$$\Phi(\lambda\hat{\varepsilon}.\bigwedge_{n=0}^{\infty} e_n(\mathsf{E}_n(\hat{\varepsilon}),\hat{\varepsilon}),\ \lambda\hat{\phi}.\bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\hat{\phi}),\hat{\phi})).$$

Before we can prove Theorem 3.17, we establish the (obvious) relationship between the partial predicates e_n and f_n and the predicates e_{n+1} and f_{n+1}. We observe that this is really an induction over the complexity of the domain **E**. As the value of n increases, the predicate e_n is able to accurately relate more of the values from the domain **E**. This is because E_n restricts the complexity of the elements we may consider at any stage. Any more complicated element, ε, is represented by the best approximation in **E** satisfying $\mathsf{E}_n(\varepsilon) = \varepsilon'$ and $\mathsf{E}_n(\varepsilon') = \varepsilon'$.

Lemma 3.18

For all $\hat{\varepsilon}$, $\hat{\phi}$ *and* $n \geq 0$

$$\begin{array}{lcl} e_n(\hat{\varepsilon}) & \Rightarrow & e_{n+1}(\mathsf{E}_n(\hat{\varepsilon}),\hat{\varepsilon}) \\ f_n(\hat{\phi}) & \Rightarrow & f_{n+1}(\mathsf{F}_n(\hat{\phi}),\hat{\phi}). \end{array}$$

Notice that it is at this point that generalizing the predicates e and f to denote equivalence, rather than approximation, would fail. If $e_n(\mathsf{E}_n(\hat{\varepsilon},\hat{\varepsilon})$ were to imply $\mathsf{E}_n(\hat{\varepsilon}) \mid \mathbf{B} = E(\hat{\varepsilon}) \mid \mathbf{B}$ then we would be unable to prove the base case, which requires that $f_1(\lambda x.\perp,\hat{\phi})$ holds for all $\hat{\phi}$. There would then be only one function allowed in the language: $\lambda x.\perp$. As Stoy remarks in a similar context, this does not define a particularly useful language.

Proof of Lemma 3.18

By induction on n.

1. Bases cases. We first observe that $f_0(\hat{\phi})$ is true, and that $\mathsf{F}_0(\hat{\phi}) = \perp$. But

$$\begin{array}{lcl} f_1(\perp,\hat{\varepsilon}) & \Leftrightarrow & \bigwedge\{e_0((\mathsf{F}_1(\perp))(\mathsf{E}_0(\hat{\alpha})),\ \mathit{Eval}(\hat{\varepsilon}_\alpha))\mid \\ & & \quad \mathit{Applied}(\hat{\varepsilon}_\alpha,\hat{\varepsilon},\hat{\alpha}) \\ & & \quad \wedge\ e_0(\mathsf{E}_0(\hat{\alpha}),\ \mathit{Eval}(\hat{\alpha}))\}, \end{array}$$

which is

$$\bigwedge\{e_0(\bot, \mathit{Eval}(\grave{e}_\alpha)) \mid \mathit{Applied}(\grave{e}_\alpha, \grave{e}, \grave{\alpha}) \\ \wedge e_0(\mathsf{E}_0(\grave{\alpha}), \mathit{Eval}(\grave{\alpha}))\},$$

But $e_0(\bot, \grave{e})$ is always true. Notice that it is here that the Lemma would break down if we wished to express equality, rather than approximation, with the predicates e and f.

Now suppose that $e_0(\acute{e})$ holds. Then by considering the cases of $\acute{e}$, we see that $e_1(\mathsf{E}_0(\acute{e}), \grave{e})$.

2. Suppose, inductively, that the Lemma is true for some value n. Then

$$f_{n+1}(\hat{e}) \Leftrightarrow \bigwedge\{e_n((\mathsf{F}_{n+1}(\acute{e}))(\mathsf{E}_n(\grave{\alpha})), \mathit{Eval}(\grave{e}_\alpha)) \mid \\ \mathit{Applied}(\grave{e}_\alpha, \grave{e}, \grave{\alpha}) \\ \wedge e_n(\mathsf{E}_n(\grave{\alpha}), \mathit{Eval}(\grave{\alpha}))\}.$$

But, by the inductive hypothesis:

$$\Rightarrow \bigwedge\{e_{n+1}((\mathsf{E}_n(\mathsf{F}_{n+1}(\acute{e}))(\mathsf{E}_n(\grave{\alpha}))), \mathit{Eval}(\grave{e}_\alpha)) \mid \\ \mathit{Applied}(\grave{e}_\alpha, \grave{e}, \grave{\alpha}) \\ \wedge e_{n+1}(\mathsf{E}_n(\mathsf{E}_n(\grave{\alpha})), \mathit{Eval}(\grave{\alpha}))\}.$$

However, $\mathsf{E}_n(\mathsf{F}_{n+1}(\acute{e})\,\mathsf{E}_n(\grave{\alpha})) = \mathsf{F}_{n+1}(\acute{e})\,\mathsf{E}_n(\grave{\alpha})$, so we have

$$f_{n+2}(\mathsf{F}_{n+1}(\acute{e}), \grave{e}).$$

Similarly,

$$\begin{array}{lll} e_{n+1}(\hat{e}) & & \\ \Leftrightarrow \quad (\grave{e} \equiv \bot) & \longrightarrow & \acute{e} \equiv \bot, \\ \quad (\grave{e} \equiv \underline{?}) & \longrightarrow & \acute{e} \sqsubseteq \underline{?}, \\ \quad (\mathit{IsBasic}(\grave{e})) & \longrightarrow & (\acute{e} \in \mathbf{B}) \wedge \acute{e} \sqsubseteq E(\grave{e}), \\ \quad (\mathit{IsFunction}(\grave{e})) & \longrightarrow & (\acute{e} \in \mathbf{F}) \wedge f_{n+1}(\mathsf{F}_{n+1}(\acute{e}), \grave{e}), \\ \quad \mathit{false}. & & \end{array}$$

Which by the above result is

$$\begin{array}{lll} \Leftrightarrow \quad (\grave{e} \equiv \bot) & \longrightarrow & \acute{e} \equiv \bot, \\ \quad (\grave{e} \equiv \underline{?}) & \longrightarrow & \acute{e} \sqsubseteq \underline{?}, \\ \quad (\mathit{IsBasic}(\grave{e})) & \longrightarrow & (\acute{e} \in \mathbf{B}) \wedge \acute{e} \sqsubseteq E(\grave{e}), \\ \quad (\mathit{IsFunction}(\grave{e})) & \longrightarrow & (\acute{e} \in \mathbf{F}) \wedge f_{n+2}(\mathsf{F}_{n+1}(\acute{e}), \grave{e}), \\ \quad \mathit{false}. & & \end{array}$$

But this is $e_{n+2}(\mathsf{E}_{n+1}(\acute{\varepsilon}), \grave{\varepsilon})$.

□

What we now wish to investigate is the relationship between e_n and e_m for particular elements, ε in **E**. Because of the projective nature of E_i, and the approximation result we have just demonstrated, we are able to conclude that the predicates are well behaved in the sense of Lemma 3.19.

Lemma 3.19

For all $\hat{\varepsilon}$, $\hat{\phi}$ *and* $n \geq 0$

$$\begin{array}{lcl} e_{n+1}(\hat{\varepsilon}) & \Rightarrow & e_n(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}) \\ f_{n+1}(\hat{\phi}) & \Rightarrow & f_n(\mathsf{F}_n(\acute{\phi}), \grave{\phi}). \end{array}$$

Proof of Lemma 3.19

By induction on n.

1. Bases cases. We first observe that $f_0(\hat{\phi})$ is true, and so trivially $f_1(\hat{\phi}) \Rightarrow f_0(\mathsf{F}_0(\acute{\phi}), \grave{\phi})$.
   Now suppose that $e_1(\hat{\varepsilon})$ holds. Then by considering the cases of $\grave{\varepsilon}$, we see that $e_0(\mathsf{E}_0(\acute{\varepsilon}), \grave{\varepsilon})$.
2. Suppose, inductively, that the Lemma is true for some value n. Then

$$\begin{array}{lcl} f_{n+2}(\hat{\varepsilon}) & \Leftrightarrow & \bigwedge\{e_{n+1}((\mathsf{F}_{n+2}(\acute{\varepsilon}))(\mathsf{E}_{n+1}(\acute{\alpha})), \mathit{Eval}(\grave{\varepsilon}_\alpha)) \mid \\ & & \quad \mathit{Applied}(\grave{\varepsilon}_\alpha, \grave{\varepsilon}, \grave{\alpha}) \\ & & \quad \wedge\, e_{n+1}(\mathsf{E}_{n+1}(\acute{\alpha}), \mathit{Eval}(\grave{\alpha}))\}. \end{array}$$

   But, by the inductive hypothesis:

$$\begin{array}{lcl} \Rightarrow & \bigwedge\{e_n((\mathsf{E}_n(\mathsf{F}_{n+2}(\acute{\varepsilon}))(\mathsf{E}_{n+1}(\acute{\alpha}))), \mathit{Eval}(\grave{\varepsilon}_\alpha)) \mid \\ & & \quad \mathit{Applied}(\grave{\varepsilon}_\alpha, \grave{\varepsilon}, \grave{\alpha}) \\ & & \quad \wedge\, e_n(\mathsf{E}_n(\acute{\alpha}), \mathit{Eval}(\grave{\alpha}))\}. \end{array}$$

   However, $\mathsf{E}_n(\mathsf{F}_{n+2}(\acute{\varepsilon})\,\mathsf{E}_{n+1}(\acute{\alpha})) = \mathsf{F}_{n+1}(\acute{\varepsilon})\,\mathsf{E}_n(\acute{\alpha})$, so we have

   $f_{n+1}(\mathsf{F}_{n+1}(\acute{\varepsilon}), \grave{\varepsilon})$.

Similarly,

$$
\begin{array}{lll}
e_{n+2}(\hat{\varepsilon}) & & \\
\Leftrightarrow \quad (\hat{\varepsilon} \equiv \bot) & \longrightarrow & \acute{\varepsilon} \equiv \bot, \\
\quad (\hat{\varepsilon} \equiv \underline{?}) & \longrightarrow & \acute{\varepsilon} \sqsubseteq \underline{?}, \\
\quad (\mathit{IsBasic}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{B}) \wedge \acute{\varepsilon} \sqsubseteq E(\hat{\varepsilon}), \\
\quad (\mathit{IsFunction}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{F}) \wedge f_{n+2}(\mathsf{F}_{n+2}(\acute{\varepsilon}), \hat{\varepsilon}), \\
\quad \mathit{false}. & &
\end{array}
$$

Which by the above result is

$$
\begin{array}{lll}
\Leftrightarrow \quad (\hat{\varepsilon} \equiv \bot) & \longrightarrow & \acute{\varepsilon} \equiv \bot, \\
\quad (\hat{\varepsilon} \equiv \underline{?}) & \longrightarrow & \acute{\varepsilon} \sqsubseteq \underline{?}, \\
\quad (\mathit{IsBasic}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{B}) \wedge \acute{\varepsilon} \sqsubseteq E(\hat{\varepsilon}), \\
\quad (\mathit{IsFunction}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{F}) \wedge f_{n+1}(\mathsf{F}_{n+1}(\acute{\varepsilon}), \hat{\varepsilon}), \\
\quad \mathit{false}. & &
\end{array}
$$

But this is $e_{n+1}(\mathsf{E}_{n+1}(\acute{\varepsilon}), \hat{\varepsilon})$.

□

We can now prove Theorem 3.17, by substituting for P and Q the values $\bigwedge_{n=0}^{\infty} e_n(\mathsf{E}(\acute{\varepsilon}), \hat{\varepsilon})$ and $\bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\acute{\phi}), \hat{\phi})$ which we hope will satisfy Φ.

Proof of Theorem 3.17

We may establish the first part by direct substitution for P and Q into the definition of Φ:

$$
\begin{array}{lll}
P(\hat{\varepsilon}) \Leftrightarrow \bigwedge_{n=0}^{\infty} \{ \ (\hat{\varepsilon} \equiv \bot) & \longrightarrow & \acute{\varepsilon} \equiv \bot, \\
\quad (\hat{\varepsilon} \equiv \underline{?}) & \longrightarrow & \acute{\varepsilon} \sqsubseteq \underline{?}, \\
\quad (\mathit{IsBasic}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{B}) \wedge \acute{\varepsilon} \sqsubseteq E(\hat{\varepsilon}), \\
\quad (\mathit{IsFunction}(\hat{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{F}) \wedge Q((\acute{\varepsilon} \mid \mathbf{F}), \hat{\varepsilon}), \\
\quad \mathit{false}\}. & &
\end{array}
$$

So we must establish that

$$
Q(\acute{\varepsilon} \mid \mathbf{F}, \hat{\varepsilon}) = \bigwedge_{n=0}^{\infty} Q(\mathsf{E}_n(\acute{\varepsilon}) \mid \mathbf{F}, \hat{\varepsilon}),
$$

which follows immediately from Lemma 3.19. We now wish to establish the second part:

$$\begin{aligned} Q(\hat{\phi}) &= \bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\phi), \dot{\phi}) \\ &= \bigwedge_{n=0}^{\infty} \bigwedge \{e_n(\mathsf{F}_n(\phi)\,\dot{\alpha}, \mathit{Eval}(\dot{\phi}_\alpha)) \mid \mathit{Applied}(\dot{\phi}_\alpha, \dot{\phi}, \dot{\alpha}) \wedge \\ &\qquad e_n(\mathsf{E}_n(\dot{\alpha}), \mathit{Eval}(\dot{\alpha}))\}. \end{aligned}$$

Now let $p_n = e_n(\mathsf{E}_n(\dot{\alpha}), \mathit{Eval}(\dot{\alpha}))$ and $q_n = e_n(\mathsf{F}_{n+1}(\phi)\,\dot{\alpha}, \mathit{Eval}(\dot{\phi}_\alpha))$. So $Q(\hat{\phi}) = \forall \hat{\alpha}\,.\,\mathit{Applied}(\dot{\phi}_\alpha, \dot{\phi}, \dot{\alpha}) \wedge \bigwedge_{n=0}^{\infty}\{p_n \Rightarrow q_n\}$. However, substituting for the left hand side of the second part we have:

$$\begin{aligned} &\bigwedge\{P(\mathsf{F}_n(\phi)\,\dot{\alpha}, \mathit{Eval}(\dot{\phi}_\alpha)) \mid \mathit{Applied}(\dot{\phi}_\alpha, \dot{\phi}, \dot{\alpha}) \wedge \\ &\qquad P(\mathsf{E}_n(\dot{\alpha}), \mathit{Eval}(\dot{\alpha}))\} \\ = \;& \forall \hat{\alpha}\,.\,\mathit{Applied}(\dot{\phi}_\alpha, \dot{\phi}, \dot{\alpha}) \wedge \{\textstyle\bigwedge_{n=0}^{\infty} p_n\} \Rightarrow \{\textstyle\bigwedge_{n=0}^{\infty} q_n\}. \end{aligned}$$

The equivalence

$$\bigwedge_{n=0}^{\infty}\{p_n \Rightarrow q_n\} = \bigwedge_{n=0}^{\infty} p_n \Rightarrow \bigwedge_{n=0}^{\infty} q_n$$

may now be established by domain induction. Suppose that for some m, $\mathsf{E}_m(\dot{\alpha}) = \dot{\alpha}$ and $\mathsf{F}_{m+1}(\phi) = \phi$. Then, by Lemma 3.19,

$$\bigwedge_{n=0}^{\infty}\{p_n \Rightarrow q_n\} = \{p_m \Rightarrow q_m\} = \{\bigwedge_{n=0}^{\infty} p_n \Rightarrow \bigwedge_{n=0}^{\infty} p_n\}.$$

Thus provided our induction principle works we have the required result. The induction principle is guaranteed by $\bigsqcup_{n=0}^{\infty} \mathsf{E}_n(\varepsilon) = \varepsilon$.

□

Notice that we immediately have Corollary 3.20.

Corollary 3.20

For all $\hat{\varepsilon}$, if $e(\hat{\varepsilon})$ and $\varepsilon \in \mathbf{B}$, then

$$\varepsilon \sqsubseteq E(\hat{\varepsilon}).$$

We are unable to deduce that $e(\hat{\varepsilon})$ implies $\acute{\varepsilon} \sqsubseteq E(\grave{\varepsilon})$, because of the following counter-example (due to Plotkin [1977]).

Let

$$f_i = \lambda g.\mathit{if}\,((g\ \mathit{True}\ \bot)\ \mathit{and}\ (g\ \bot\ \mathit{True})\mathit{and}\ (\mathit{not}\,(g\ \mathit{False}\ \mathit{False})))i\bot,$$

where *and* and *not* are defined as usual. Informally, $f_i\,g$ is bottom, unless g is parallel-or, in which case $f_i\,g$ is i. In the operational semantics we have no way to represent parallel-or and so, for all i and j,

$$\bot = \grave{f}_i\,\grave{g} = \grave{f}_j\,\grave{g}.$$

However, in the denotational semantics, $\acute{f}_1$ and $\acute{f}_2$ are distinct elements of **E**. This is because parallel-or is certainly an element of **E**, and therefore $\acute{f}_1$ and $\acute{f}_2$ must be distinguishable when applied to parallel-or. If the predicate e were able to express the congruence of functions, we would be able to generate the contradictory result that $\acute{f}_1 = \acute{f}_2$.

To conclude this section, we have defined a predicate e to compare results in the two semantics we are investigating. This predicate is well defined by construction. The reader may wonder why we went to the trouble of defining e and f, when we could have used the alternatives $\bigwedge_{n=0}^{\infty} e_n(\mathsf{E}(\acute{\varepsilon}), \grave{\varepsilon})$ and $\bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\acute{\phi}), \grave{\phi})$. The reason is that in the next section we would have to have shown this equivalence anyway, as the alternative predicates are too cumbersome to use directly. We now make use of the predicates e and f to investigate the partial congruence of our two semantics.

3.4 Analysis of the Denotational Semantics

Finally, we wish to show that Theorem 3.21 holds.

Theorem 3.21

For all Π *in* Prog,

$$e(\acute{\mathcal{P}}\,[\![\Pi]\!],\ \grave{\mathcal{P}}\,[\![\Pi]\!]).$$

We notice that by Corollary 3.20, this will imply the partial congruence $\acute{\mathcal{P}}\,[\![\Pi]\!] \sqsubseteq E(\grave{\mathcal{P}}\,[\![\Pi]\!])$, of Theorem 3.12, provided $\acute{\mathcal{P}}\,[\![\Pi]\!] \in \mathbf{B}$. Before proving Theorem 3.21, we state and prove a lemma that will be useful in the main theorem. It is a general observation about the nature of application in the two definitions of the language. It allows us to prove easily all of the structural inductions that involve the $\mathrm{E} = [\![\mathrm{E}_0\,(\mathrm{E}_1)]\!]$ case.

Lemma 3.22

Suppose that $\gamma\,\ell \in \grave{\mathbf{A}}$ *and that* $\gamma\,\ell \mid \grave{\mathbf{A}} = (\ell_0, \ell_1)$. *Suppose further that* $e(\acute{e}_i)$ *holds for* $i = 0, 1$, *where*

$$\grave{e}_i = \mathit{Eval}((\gamma, \ell_i), \delta).$$

Then $e(\acute{e})$, *where*

$$\begin{array}{rcl} \acute{e} & = & (\acute{e}_0 \in \acute{\mathbf{F}}) \longrightarrow (\acute{e}_0 \mid \acute{\mathbf{F}})\,\acute{e}_1, \underline{\underline{?}} \\ \textit{and} & & \\ \grave{e} & = & \mathit{IsFunction}(\grave{e}_0) \longrightarrow \mathit{Eval}((\gamma, \ell), \delta), \\ & & (\mathit{IsBasic}(\grave{e}_0) \vee (\grave{e}_0 = \underline{\underline{?}})) \longrightarrow \underline{\underline{?}}, \perp. \end{array}$$

Put another way, suppose that we have two subgraphs with roots ℓ_0 and ℓ_1, and that $\acute{e}_0$ and $\acute{e}_1$ approximate the reduced graphs. Then Lemma 3.22 states that the result of reducing the application of these two graphs (which we represent as $\mathit{Eval}((\gamma, \ell), \delta)$) will be approximated by the value obtained by applying the two values $\acute{e}_0$ and $\acute{e}_1$.

This is what we would expect to happen.

Proof of Lemma 3.22

Consider the sequence of states σ_n and σ'_n, where

$$\sigma_n = \mathit{Step}^n((\gamma, \ell), \delta) \text{ and } \sigma'_n = \mathit{Step}^n((\gamma, \ell_0), \delta).$$

We now prove that the following relation holds between elements of these two reduction sequences.

If $\sigma'_n = ((\gamma_n, \ell_0), \delta)$ then $\sigma_n = ((\gamma_n, \ell), \delta)$.

This is proved by induction on n.

We see immediately that

$$\mathit{Spine}(\gamma_n, \ell) = \mathit{Spine}(\gamma_n, \ell_0) +\!\!+ [\mathit{Elide}(\ell)].$$

And hence that $\mathit{not}(\mathit{Done}(\sigma'_n))$ implies $\mathit{not}(\mathit{Done}(\sigma_n))$, (although the converse is not necessarily true).

Now consider the four values for $\grave{e}_0$.

1. $\dot{e}_0 \equiv \perp$
   If $\dot{e}_0 \equiv \perp$, then either $Spine(\gamma_n, \ell_0) = \perp$ for all i greater than some m, or there is no m such that $Done(\sigma'_m)$ holds.
   In the first case we see that $Spine(\gamma_i, \ell) = \perp$ for all i greater than m, as well.
   In the second, our second observation shows that there can be no terminal state satisfying $Done(\sigma_m)$.
2. $\dot{e}_0 \equiv \stackrel{?}{=}$, $IsBasic(\dot{e}_0)$ and $IsFunction(\dot{e}_0)$
   In these cases, suppose that $\dot{e}_0 = ((\gamma, \ell_0), \delta) = \sigma'_n$ for some n. But in this case $\dot{e} = Eval((\gamma, \ell), \delta)$, from the above relation on states.

□

We may now establish Theorem 3.21 by structural induction.

Proof of Theorem 3.21

Substituting for $\dot{\mathcal{P}}$, $\dot{\mathcal{P}}$ and Π, we have:

For all E in Exp and all Δ in Defs:

$$e(\dot{\mathcal{E}}\,[\![\mathrm{E}]\!]\,fix(\dot{\mathcal{D}}\,[\![\Delta]\!]),\ Eval(\dot{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\{\}\,\sigma_{\mathrm{init}},\ \dot{\mathcal{D}}\,[\![\Delta]\!])).$$

This is proved by structural induction on the expression E.

1. $\mathrm{E} = [\![\mathrm{I}]\!]$
   Firstly $\dot{\mathcal{E}}\,[\![\mathrm{I}]\!]\,fix(\dot{\mathcal{D}}\,[\![\Delta]\!]) = fix(\dot{\mathcal{D}}\,[\![\Delta]\!])\,[\![\mathrm{I}]\!]$. Next
   $$E(\dot{\mathcal{E}}\,[\![\mathrm{I}]\!]\,\{\}\,\sigma_{\mathrm{init}},\ \dot{\mathcal{D}}\,[\![\Delta]\!]) = acute(\dot{\mathcal{D}}\,[\![\Delta]\!])\,[\![\mathrm{I}]\!],$$
   But, by Lemma 3.2, $fix(\dot{\mathcal{D}}\,[\![\Delta]\!]) = acute(\dot{\mathcal{D}}\,[\![\Delta]\!])$.
2. $\mathrm{E} = [\![\mathrm{B}]\!]$
   Let $\dot{e} = Eval(\dot{\mathcal{E}}\,[\![\mathrm{B}]\!]\,\{\}\,\sigma_{\mathrm{init}},\ [\![\Delta]\!])$. Then $IsBasic(\dot{e})$ and $E(\dot{e})$ equals $\mathcal{B}\,[\![\mathrm{B}]\!]$ in É. But $\dot{\mathcal{E}}\,[\![\mathrm{B}]\!]\,fix(\dot{\mathcal{D}}\,[\![\Delta]\!]) = \mathcal{B}\,[\![\mathrm{B}]\!]$ in É.
3. $\mathrm{E} = [\![\mathrm{E}_0\,(\mathrm{E}_1)]\!]$
   By Inductive Hypothesis, and Lemma 3.22.

□

We have now proved Theorem 3.12 by the obvious application of Corollary 3.20 to Theorem 3.21. With the completion of the proof of this theorem, we have proved the main result of this chapter – the complete congruence of Theorem 3.3. Schmidt refers to an operational semantics which is completely congruent to a denotational semantics as both faithful and terminating with respect to the denotational semantics [Schmidt 86, Section 10.7].

We summarize the route that has been taken to establish congruence. First the proof broke down into two separate parts: Theorem 3.4 which says that the operational semantics approximates the denotational semantics and Theorem 3.12 which says that the denotational semantics approximates the operational semantics.

To prove Theorem 3.4 we establish that a single reduction step does not change the meaning of the graph (Theorem 3.7) and from that used fixpoint induction to deduce that the meaning of the initial graph was approximated by that of the final one. A structural induction was sufficient to show that the initial graph was congruent to the denotational semantics.

To prove Theorem 3.12 we first establish that the predicates required to express the partial congruence exist. This is the result of Section 3.3. In Section 3.4 we used structural induction to show that $e(\dot{\mathcal{P}}[\![\Pi]\!], \dot{\mathcal{P}}[\![\Pi]\!])$ holds for all programs Π.

3.5 Related Work

The use of fixpoint induction to prove partial congruence follows immediately from the work of Manna, Ness and Vuillemin [1973] on fixpoint properties. The technique of structural induction, which we have used repeatedly, was first described by Burstall in [1969].

The use of inclusive predicates was first described by Milne in [1974] to describe mode declarations in Algol 60. With Strachey, he used this process again to prove compiler correctness in [Milne and Strachey 76]. Reynolds has used directed complete predicates in a similar way in [1974] and Gordon uses a related technique in [1973]. Further use of the inclusive predicate strategy is made in [Stoy 77, Chapter 13] and [Stoy 81].

An alternative domain construction for $\mathbf{E}$, related to Scott's $\mathbf{D}_\infty$ model [1981], can be used to prove that the inclusive predicates e and f exist. The relationship between the $\mathbf{D}_\infty$ and $\mathcal{P}\omega$ models of the pure λ-calculus is discussed in [Wadsworth 76]. In [1976], Scott presents relationships between projection functions and retracts.

The full abstraction problem for the typed lambda calculus was first discovered by Plotkin [1977]. His solution was to extend the language with which he was working, so that it included parallel-or. Other approaches have been undertaken, by Milner [1977] and Mulmuley [1984, 1986] amongst others. These have been restrictive, in that the domain **E** includes only sequentially implementable functions. Ong [1988] gives a fully abstract model of the lazy lambda calculus, using bisimulation logical relations.

A similar result to that obtained in this chapter has been derived by Klop [1980]. His work on combinatory reduction systems is based on a syntactic model of reduction systems, rather than the domain based model investigated in this thesis.

3.6 Conclusion

We have now shown that the operational semantics we have defined in Chapter 2, correctly implements the standard denotational semantics. There are a number of other ways to state this result. We may regard the operational semantics as a graph rewriting system, in which case we have provided a termination result; alternatively, we may regard the congruence proof as a validation of the weak head normal reduction mechanism of Peyton Jones [1987]. Stated simply: graph reduction implements functional languages correctly.

One worry we have is that this language has no built-in functions or data structuring facilities. We address this problem in the next chapter.

Chapter 4

Extending The Language

In this chapter we extend the result of Chapter 3 to cover a language with structured data and built-in functions. We consider various ways to represent data structures and select the simplest. We also implement a representative sample of built-in functions.

4.1 Extended Denotational Semantics

The first question we face is whether we intend to model a typed or untyped language. The majority of modern functional languages have strong polymorphic typing, but to introduce this into our language we would need to provide a type-checking function $\mathcal{P}_T$: Prog $\rightarrow$ T. Such a function would be used to provide a denotational semantics along the lines

$$\mathcal{P}_T[\![\Pi]\!] \longrightarrow \mathcal{P}[\![\Pi]\!], \underline{?}$$

The problem with such an approach is that to typecheck Π we need to carry out a dependency analysis. In this analysis we transform the program so that all mutually recursive definitions are really mutually recursive. Failure to do this may make it impossible to type-check a program, see [Mycroft 84].

As such source-level translations are against the spirit of denotational semantics, we now consider an alternative definition of programs. In this alternative we allow general local definitions in any expression, subject only to the criterion that mutually recursive definitions must satisfy our dependency criterion. The problem with this approach is that our operational semantics must incorporate a lambda-lifter or similar mechanism to translate back to

ε	$\in$	$\mathbf{E}$	$=$	$\mathbf{F}+\mathbf{C}+\mathbf{B}$	(Expression Values)
ϕ	$\in$	$\mathbf{F}$	$=$	$[\mathbf{E}\to\mathbf{E}]$	(Function Values)
χ	$\in$	$\mathbf{C}$	$=$	$\mathbf{E}\times\mathbf{E}$	(Constructors)
β	$\in$	$\mathbf{B}$			(Primitive Values)
ρ	$\in$	$\mathbf{U}$	$=$	$[\mathrm{Ide}\to\mathbf{E}]$	(Environments)

Figure 4.1: Value Domains for Denotational Semantics (Extended for Constructors)

the form we currently have for programs. We note that as a result of this, our operational semantics is indifferent to the existence of a typing scheme.

The conclusion of this discussion is that for simplicity we shall consider an untyped language. After all, we can translate a typed language into our untyped language.

We now move on to consider how we wish to represent constructors. The alternatives are to have a single constructor akin to the LISP CONS, or to extend the language to deal with arbitrary constructors. The example of LISP shows that a single constructor is sufficient for programming purposes. The G-machine constructors may be represented, at least in abstract, as elements of the domain $\mathbf{C} = \mathrm{Ide}\times\mathbf{L}^*$, where the identifier models the "tag" and each $\ell\in\mathbf{L}$ represents the pointers into the heap for each argument. The problem with this approach is that we need a number of selectors for each constructor along with a tag testing function. For this reason we consider a single constructor cons, with selectors head and tail and a tag testing function null and constant nil. As tail is very similar to head we will omit it from further discussion.

We next consider which arithmetic, comparison and conditional functions we require. We shall presume that arithmetic is defined over some suitable representation of the integers or a subset of them. This domain will be called $\mathbf{Z}$. The comparison operations are defined only over $\mathbf{Z}$. Structural equality will have to be defined separately. Finally, we have a single conditional function, if, which behaves in the appropriate way on the domain T. We can summarise this information by providing the standard semantics for such operations in Figures 4.1 and 4.2.

We will also allow expressions to construct cyclic graphs, although we note that as far as the denotational semantics is concerned we do not have any notion of cyclic definitions.

$$
\begin{array}{lcl}
\rho_{\text{basic}}[\![\text{null}]\!] & = & \lambda\varepsilon.(\varepsilon = \text{nil}) \\
\rho_{\text{basic}}[\![\text{head}]\!] & = & \lambda\varepsilon.(\varepsilon \in \mathbf{C}) \longrightarrow \mathit{fst}(\varepsilon \mid \mathbf{C}), \underline{?} \\
\rho_{\text{basic}}[\![\text{cons}]\!] & = & \lambda\varepsilon_0\lambda\varepsilon_1.(\varepsilon_0, \varepsilon_1)\ \mathbf{in}\ \mathbf{E} \\
\rho_{\text{basic}}[\![+]\!] & = & \lambda\varepsilon_0\lambda\varepsilon_1.(\varepsilon_0 \in \mathbf{Z}) \longrightarrow \\
 & & \quad ((\varepsilon_1 \in \mathbf{Z}) \longrightarrow (\varepsilon_0 \mid \mathbf{Z} + \varepsilon_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{E}, \underline{?}), \underline{?} \\
\rho_{\text{basic}}[\![=]\!] & = & \lambda\varepsilon_0\lambda\varepsilon_1.(\varepsilon_0 \in \mathbf{Z}) \longrightarrow \\
 & & \quad ((\varepsilon_1 \in \mathbf{Z}) \longrightarrow (\varepsilon_0 \mid \mathbf{Z} = \varepsilon_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{E}, \underline{?}), \underline{?} \\
\rho_{\text{basic}}[\![\text{if}]\!] & = & \lambda\varepsilon_0\lambda\varepsilon_1\lambda\varepsilon_2.(\varepsilon_0 \in \mathbf{T}) \longrightarrow ((\varepsilon_0 \mid \mathbf{T}) \longrightarrow \varepsilon_1,\ \varepsilon_2), \underline{?}
\end{array}
$$

Figure 4.2: The initial environment ρ_{basic}

σ	$\in$	$\mathbf{S}$	$=$	$(\mathbf{G} \times \mathbf{L}) \times \mathbf{D}$	(States)
γ	$\in$	$\mathbf{G}$	$=$	$[\mathbf{L} \to \mathbf{N}]$	(Graphs)
ν	$\in$	$\mathbf{N}$	$=$	$\mathbf{A} + \mathbf{I} + \mathbf{B} + \text{Ide} + \mathbf{C}$	(Nodes)
		$\mathbf{A}$	$=$	$\mathbf{L} \times \mathbf{L}$	(Application Nodes)
		$\mathbf{I}$	$=$	$\mathbf{L}$	(Indirection Nodes)
χ	$\in$	$\mathbf{C}$	$=$	$\mathbf{L} \times \mathbf{L}$	(Constructor Nodes)
β	$\in$	$\mathbf{B}$			(Basic Values)
ρ	$\in$	$\mathbf{U}$	$=$	$[\text{Ide} \to \mathbf{L}]$	(Local Environments)
δ	$\in$	$\mathbf{D}$	$=$	$[\text{Ide} \to \text{Comb}]$	(Global Environments)
ℓ	$\in$	$\mathbf{L}$			(Node Labels)
ϕ	$\in$	$\mathbf{L}^*$			(Spines)
κ	$\in$	$\mathbf{K}$	$=$	$[(\mathbf{G} \times \mathbf{L}) \to (\mathbf{G} \times \mathbf{L})]$	(Continuations)

Figure 4.3: Value Domains for Operational Semantics (Extended for Constructors)

We note that we now wish to make explicit the basic value domain **B**, as $\mathbf{B} = \mathbf{Z} + \mathbf{T} + \{\text{nil}\}$. We will dispense with projections, injections and membership tests associated with **B** as they can clearly be inferred from the context.

The resulting language thus resembles a lambda-lifted form of LispKit, which is described in [Henderson 80] and [Henderson *et al.* 82].

4.2 Extended Operational Semantics

The operational semantics of our extended language is now defined. In

$$
\begin{array}{lcl}
Dump(\sigma) & = & Done(\sigma) \longrightarrow [\ell : \phi], \\
 & & (1 = [\![\text{null}]\!]) \longrightarrow Dump_{\text{Null}}(\ell : \phi)\,\sigma, \\
 & & (1 = [\![\text{head}]\!]) \longrightarrow Dump_{\text{Head}}(\ell : \phi)\,\sigma, \\
 & & (1 = [\![\text{cons}]\!]) \longrightarrow Dump_{\text{Cons}}(\ell : \phi)\,\sigma, \\
 & & (1 = [\![+]\!]) \longrightarrow Dump_{\text{Plus}}(\ell : \phi)\,\sigma, \\
 & & (1 = [\![=]\!]) \longrightarrow Dump_{\text{Eq}}\,\phi)\,\sigma, \\
 & & (1 = [\![\text{if}]\!]) \longrightarrow Dump_{\text{If}}(\ell : \phi)\,\sigma, \\
 & & \qquad [\ell : \phi] \\
 & & \text{where } (\ell : \phi) = Spine(\gamma, \tau) \\
 & & \qquad ((\gamma, \tau), \delta) = \sigma \\
 & & \qquad 1 = (\gamma\,\ell) \mid \text{Ide} \\
Dump_{\text{Null}}\,\phi((\gamma, \tau), \delta) & = & (Done(\sigma_{a_1}) \longrightarrow [], Dump(\sigma_{a_1})) +\!\!+ [\phi] \\
 & & \quad \text{where } \sigma_{a_1} = ((\gamma, Arg(\gamma, (\phi\ !\ 1))), \delta) \\
Dump_{\text{Head}}\,\phi((\gamma, \tau), \delta) & = & (Done(\sigma_{a_1}) \longrightarrow [], Dump(\sigma_{a_1})) +\!\!+ [\phi] \\
 & & \quad \text{where } \sigma_{a_1} = ((\gamma, Arg(\gamma, (\phi\ !\ 1))), \delta) \\
Dump_{\text{Cons}}\,\phi((\gamma, \tau), \delta) & = & [\phi] \\
Dump_{\text{Plus}}\,\phi((\gamma, \tau), \delta) & = & (Done(\sigma_{a_1}) \longrightarrow (Done(\sigma_{a_2}) \longrightarrow [], \\
 & & \quad Dump(\sigma_{a_2})), Dump(\sigma_{a_1})) +\!\!+ [\phi] \\
 & & \quad \text{where } \sigma_{a_i} = ((\gamma, Arg(\gamma, (\phi\ !\ i))), \delta) \\
Dump_{\text{Eq}}\,\phi((\gamma, \tau), \delta) & = & (Done(\sigma_{a_1}) \longrightarrow (Done(\sigma_{a_2}) \longrightarrow [], \\
 & & \quad Dump(\sigma_{a_2})), Dump(\sigma_{a_1})) +\!\!+ [\phi] \\
 & & \quad \text{where } \sigma_{a_i} = ((\gamma, Arg(\gamma, (\phi\ !\ i))), \delta) \\
Dump_{\text{If}}\,\phi((\gamma, \tau), \delta) & = & (Done(\sigma_{a_1}) \longrightarrow [], Dump(\sigma_{a_1})) +\!\!+ [\phi] \\
 & & \quad \text{where } \sigma_{a_1} = ((\gamma, Arg(\gamma, (\phi\ !\ 1))), \delta)
\end{array}
$$

Figure 4.4: The *Dump* and $Dump_*$ functions

Figure 4.3, we redefine the domain of nodes, **N**, to include a constructor type. This is represented by **C**. We let $\mathbf{B} = \mathbf{Z} + \mathbf{T} + \{nil\}$, and as we did in the denotational semantics, we will infer from the context the various projections, injections and membership tests associated with **B**.

The principal difference between the operational semantics of Chapter 2 and the one presented here is that we now work with a list of spines, rather than a single spine.

In this thesis we will refer to such a structure as a dump. This terminol-

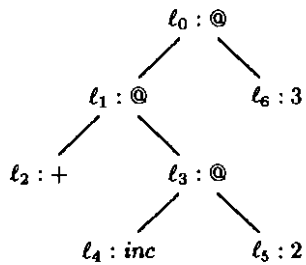

Figure 4.5: Example of a Graph

$$
\begin{array}{lcll}
Step(\sigma) & = & (\mathrm{I} = [\![\mathrm{null}]\!]) & \longrightarrow \; NullStep\,\phi\,\sigma, \\
 & & (\mathrm{I} = [\![\mathrm{head}]\!]) & \longrightarrow \; HeadStep\,\phi\,\sigma, \\
 & & (\mathrm{I} = [\![\mathrm{cons}]\!]) & \longrightarrow \; ConsStep\,\phi\,\sigma, \\
 & & (\mathrm{I} = [\![+]\!]) & \longrightarrow \; PlusStep\,\phi\,\sigma, \\
 & & (\mathrm{I} = [\![=]\!]) & \longrightarrow \; EqStep\,\phi\,\sigma, \\
 & & (\mathrm{I} = [\![\mathrm{if}]\!]) & \longrightarrow \; IfStep\,\phi\,\sigma, \\
 & & & \quad OtherStep\,(\delta\,[\![\mathrm{I}]\!])\,\phi\,\sigma \\
 & \text{where} & (\ell : \phi) : \psi & = \; Dump\,\sigma \\
 & & ((\gamma, r), \delta) & = \; \sigma \\
 & & \mathrm{I} & = \; (\gamma\,\ell) \mid \mathrm{Ide}
\end{array}
$$

Figure 4.6: The *Step* function

ogy conflicts with that of the G-machine papers [Johnsson 83], [Johnsson 84] and [Johnsson 87], where a dump is a list of pairs, each pair being a continuation and a spine.

In Figure 4.4 we construct the dump of a state by first determining the spine of the state. If this is a built-in function ***and it has enough arguments***[1],

[1]It may be possible to ignore this condition and reduce strict arguments immediately. Such a scheme is in general incorrect; the conditions under which it can be fudged are beyond the scope of this thesis. This form of reduction was proposed by Peyton Jones and

$$
\begin{array}{lcl}
\mathit{NullStep}(\ell : \phi)\,\sigma & = & \mathit{Update}\,\ell\,\nu\,\sigma \\
 & & \text{where } \nu = (\mathit{Value}\,\ell\,\sigma) = \mathit{nil}\ \mathbf{in}\ \mathbf{N} \\
\mathit{HeadStep}(\ell : \phi)\,\sigma & = & \nu \in \mathbf{C} \longrightarrow \mathit{Update}\,\ell\,(\mathit{fst}(\nu))\,\sigma, \stackrel{?}{=} \\
 & & \text{where } \nu = \mathit{Value}\,\ell\,\sigma \\
\mathit{ConsStep}(\ell_0 : \ell_1 : \phi)\,\sigma & = & \mathit{Update}\,\ell_1\,((\ell_0', \ell_1')\ \mathbf{in}\ \mathbf{N})\,\sigma \\
 & & \text{where } ((\gamma, r), \delta) = \sigma \\
 & & \qquad \ell_i' = \mathit{Arg}(\gamma, \ell_i) \\
\mathit{PlusStep}(\ell_0 : \ell_1 : \phi)\,\sigma & = & \nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z} \longrightarrow \mathit{Update}\,\ell_1\,\nu\,\sigma, \stackrel{?}{=} \\
 & & \text{where } \nu_i = \mathit{Value}\,\ell_i\,\sigma \\
 & & \qquad \nu = \nu_0 \mid \mathbf{Z} + \nu_1 \mid \mathbf{Z} \\
\mathit{EqStep}(\ell_0 : \ell_1 : \phi)\,\sigma & = & \nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z} \longrightarrow \mathit{Update}\,\ell_1\,\nu\,\sigma, \stackrel{?}{=} \\
 & & \text{where } \nu_i = \mathit{Value}\,\ell_i\,\sigma \\
 & & \qquad \nu = \nu_0 \mid \mathbf{Z} = \nu_1 \mid \mathbf{Z} \\
\mathit{IfStep}(\ell_0 : \ell_1 : \ell_2 : \phi)\,\sigma & = & \nu \in \mathbf{T} \longrightarrow \mathit{Update}\,\ell\,(\ell'\ \mathbf{in}\ \mathbf{N})\,\sigma, \stackrel{?}{=} \\
 & & \text{where } \ell' = (\mathit{Value}\,\ell_0\,\sigma) \longrightarrow \ell_1, \ell_2 \\
\mathit{OtherStep}\,[\![\Gamma]\!]\,\phi((\gamma, \ell), \delta) & = & \mathit{Update}\,\ell_r\,(\ell'\ \mathbf{in}\ \mathbf{N})((\gamma', \ell), \delta) \\
 & & \text{where } (\gamma', \ell') = \mathcal{C}\,[\![\Gamma]\!]\,\rho_{\mathit{arid}}\,\phi\,(\gamma, \ell) \\
 & & \qquad \ell_r = \phi\,!\,(\mathit{Args}\,[\![\Gamma]\!] - 1)
\end{array}
$$

Figure 4.7: The Built-in Reductions: $*Step$

$$
\begin{array}{lcl}
\mathit{Update}\,\ell\,\nu\,((\gamma, r), \delta) & = & ((\gamma \oplus \{\ell \mapsto \nu\}, r), \delta) \\
\mathit{Value}\,r\,((\gamma, \ell), \delta) & = & \gamma\,(\mathit{Elide}\,\gamma\,(\mathit{Arg}(\gamma, r)))
\end{array}
$$

Figure 4.8: Auxiliary Functions for $*Step$ Functions

we will test the strict arguments to see whether their evaluation has been completed. If there is a strict argument that has not been evaluated, we construct a dump of that argument and append the singleton list of the spine. We illustrate this in Figure 4.5. The dump of this state will be $[[\ell_4, \ell_3], [\ell_2, \ell_1, \ell_0]]$.

After determining the dump, we are able to perform a reduction step. This is performed by the function *Step* defined in Figure 4.6. This will operate only on the top element of the dump and we will refer to this as the *head* of the dump or the *redex spine*. If a built-in function is at the

is intermediate in power between his weak head normal reducer and Wadsworth's head normal reducer.

head of the redex spine, then its strict arguments must be evaluated and the reduction of the built-in function can proceed on that basis. If the function at the head of the redex spine is not a built-in function then the reduction proceeds in the same way that it did in the original operational semantics of Chapter 2.

4.3 Congruence for the Extended Language

In the previous sections we have amended the denotational semantics and operational semantics to realise constructor and other built-in operations. The proof that these definitions of the language are congruent follows the same pattern as that of Chapter 3. The strategy we used then is now repeated and therefore we repeat Figure 3.1. As the last congruence proof of Chapter 3 was presented in considerable detail, we will omit some of the detail in the congruence proof of this chapter, and only prove those theorems that have changed.

In order to establish congruence we must again use the technique of Chapter 3. First we redefine *acute* so that the identifiers corresponding to the built-in functions are defined to have the same derepresentation value as that given by the denotational semantics. From this we establish that Lemma 3.2 still holds and this establishes that the two environments are congruent. Now, because the built-in reduction steps in Figure 4.7, satisfy Theorem 4.1, they do not change the meaning of the derepresented graph.

Theorem 4.1

For all σ *in* $\grave{\mathrm{S}}$, *such that* $Done(\sigma)$ *does not hold:*

$$E(\sigma) \sqsupseteq E(Step(\sigma)).$$

We are therefore able to deduce that each reduction step does not change the meaning of the graph thus establishing a result like Theorem 3.7. From this we are able to deduce by fixpoint induction that the interpreter approximates the denotational result; this corresponds to Theorem 3.4.

To prove that the denotational result approximates that of the operational semantics (Theorem 3.12) we are required to re-prove that the new predicates exist for the new domain $\acute{\mathrm{E}}$. When this is proved, because we have established that Lemma 3.2 holds for the new definition of *acute*, we

Theorem 3.3 This proves that $\dot{\mathcal{P}}[\![\Pi]\!] = E(\dot{\mathcal{P}}[\![\Pi]\!])$, provided that $\dot{\mathcal{P}}[\![\Pi]\!] \in \mathbf{B}$. It is established by fixpoint induction on the state. It requires

Theorem 3.4 This proves that $\dot{\mathcal{P}}[\![\Pi]\!] \sqsupseteq E(\dot{\mathcal{P}}[\![\Pi]\!])$. It requires

Theorem 3.7 This proves that an individual reduction step on a state will produce a new state that approximates the old one, under derepresentation. It requires

Lemma 3.8 Indirection Node Equivalence Lemma.

Lemma 3.9 Spine Derepresentation Lemma.

Lemma 3.10 Combinator Substitution Lemma.

Theorem 3.12 This proves that $\dot{\mathcal{P}}[\![\Pi]\!] \sqsubseteq E(\dot{\mathcal{P}}[\![\Pi]\!])$, provided $\dot{\mathcal{P}}[\![\Pi]\!] \in \mathbf{B}$. It is established by structural induction on the program, using inclusive predicates. It requires

Lemma 3.2 Proves that derepresentation of a combinator name in the operational semantics is the same value as that supplied by the denotational semantics, for a given syntactic structure Δ.

Theorem 3.17 This proves that the predicates e and f exist. It requires

Lemma 3.18 Partial Predicate Projection Lemma.

Lemma 3.19 Partial Predicate Injection Lemma.

Corollary 3.20 Proves that the predicates establish the required approximation.

Theorem 3.21 Proves that all initial expressions satisfy the predicates. It requires

Lemma 3.22 Proves that the predicates imply the correct behaviour for application nodes.

Figure 3.1: Outline of the Proof Structure

have proved that the denotational result approximates the operational (Theorem 3.12). Notice that the language bas not changed, only the initial environment in which programs are executed.

We now redefine the predicates e and f to incorporate c.

Definition 4.2

$$
\begin{array}{lcll}
e(\hat{\varepsilon}) & \Leftrightarrow & (\hat{\varepsilon} \equiv \bot) & \longrightarrow \ \acute{\varepsilon} \equiv \bot, \\
 & & (\hat{\varepsilon} \equiv \underline{?}) & \longrightarrow \ \acute{\varepsilon} \sqsubseteq \underline{?}, \\
 & & (\mathit{IsBasic}(\hat{\varepsilon})) & \longrightarrow \ (\acute{\varepsilon} \in \mathbf{B}) \wedge \acute{\varepsilon} \sqsubseteq \mathit{E}(\hat{\varepsilon}), \\
 & & (\mathit{IsFunction}(\hat{\varepsilon})) & \longrightarrow \ (\acute{\varepsilon} \in \mathbf{F}) \wedge f((\acute{\varepsilon} \mid \mathbf{F}), \hat{\varepsilon}), \\
 & & (\mathit{IsCons}(\hat{\varepsilon})) & \longrightarrow \ (\acute{\varepsilon} \in \dot{\mathbf{C}}) \wedge c((\acute{\varepsilon} \mid \dot{\mathbf{C}}), \hat{\varepsilon}), \\
 & & \mathit{false} &
\end{array}
$$

$$
f(\hat{\varepsilon}) \Leftrightarrow \bigwedge\{e(\acute{\varepsilon}(\hat{\alpha}), \mathit{Eval}(\hat{\varepsilon}_\alpha)) \mid \mathit{Applied}(\hat{\varepsilon}_\alpha, \hat{\varepsilon}, \hat{\alpha}) \wedge e(\hat{\alpha}, \mathit{Eval}(\hat{\alpha}))\}
$$

$$
\begin{array}{lcl}
c(\hat{\chi}) \Leftrightarrow (\hat{\chi} \equiv \bot) \vee (e(\hat{\varepsilon}_0) \wedge s(\hat{\varepsilon}_1)) & & \\
\quad \text{where} \ ((\gamma, \ell), \delta) & = & \hat{\chi} \\
\quad (\acute{\varepsilon}_0, \acute{\varepsilon}_1) & = & \acute{\chi} \\
\quad (\dot{\ell}_0, \dot{\ell}_1) & = & \gamma(\mathit{Elide}(\gamma, \ell)) \mid \dot{\mathbf{C}}
\end{array}
$$

$$
\begin{array}{l}
\mathit{IsCons}((\gamma, \ell), \delta) = \\
\qquad (\mathit{Spine}(\gamma, \ell) = [\ell]) \wedge (\gamma\, \ell \in \dot{\mathbf{C}})
\end{array}
$$

The predicates we require may not exist. To prove that they do we must use induction on the complexity of the domains that they are specified on. This time the domain $\mathbf{E}$ is specified as the range of the retraction E where $\mathsf{E} = \mathit{fix}(\lambda \mathsf{E}.((\mathsf{E} \multimap \mathsf{E}) \oplus (\mathsf{E} \otimes \mathsf{E}) \oplus \mathsf{B}))$. We may define E so that numeric induction is valid by using E_n, F_n and C_n.

$$
\begin{array}{lcl}
\mathsf{E}_n & = & \mathsf{F}_n \oplus \mathsf{C}_n \oplus \mathsf{B} \\
\mathsf{F}_0 & = & \bot \\
\mathsf{F}_{n+1} & = & \mathsf{E}_n \multimap \mathsf{E}_n \\
\mathsf{C}_0 & = & \bot \otimes \bot \\
\mathsf{C}_{n+1} & = & \mathsf{E}_n \otimes \mathsf{E}_n
\end{array}
$$

The partial predicates e_n, f_n and c_n are defined on the sub domains $\mathsf{E}_n(\varepsilon)$ for $\varepsilon \in \mathbf{E}$, $\mathsf{F}_n(\phi)$ for $\phi \in \mathbf{F}$ and $\mathsf{C}_n(\chi)$ for $\chi \in \mathbf{C}$.

They are defined in the obvious way *c.f.* Chapter 3, Definition 3.15. Again the following conditions may be derived from them.

Lemma 4.3

For all $\hat{\varepsilon}$, $\hat{\phi}$, $\hat{\chi}$ *and* $n \geq 0$

$$
\begin{array}{lcl}
e_n(\hat{\varepsilon}) & \Rightarrow & e_{n+1}(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}) \\
e_{n+1}(\hat{\varepsilon}) & \Rightarrow & e_n(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}) \\
 & & \\
f_n(\hat{\phi}) & \Rightarrow & f_{n+1}(\mathsf{F}_n(\acute{\phi}), \grave{\phi}) \\
f_{n+1}(\hat{\phi}) & \Rightarrow & f_n(\mathsf{F}_n(\acute{\phi}), \grave{\phi}) \\
 & & \\
c_n(\hat{\chi}) & \Rightarrow & c_{n+1}(\mathsf{C}_n(\acute{\chi}), \grave{\chi}) \\
c_{n+1}(\hat{\chi}) & \Rightarrow & c_n(\mathsf{C}_n(\acute{\chi}), \grave{\chi}).
\end{array}
$$

From this we may immediately conclude Corollary 4.4.

Corollary 4.4

For all $\hat{\varepsilon}$, $\hat{\phi}$, $\hat{\chi}$ *and* $n \geq 0$

$$
\begin{array}{lcl}
e_n(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}) & \Rightarrow & \begin{cases} e_i(\mathsf{E}_i(\acute{\varepsilon}), \grave{\varepsilon}), & \text{if } i \leq n \\ e_i(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}), & \text{if } i \geq n \end{cases} \\
 & & \\
f_n(\mathsf{F}_n(\acute{\phi}), \grave{\phi}) & \Rightarrow & \begin{cases} f_i(\mathsf{F}_i(\acute{\phi}), \grave{\phi}), & \text{if } i \leq n \\ f_i(\mathsf{F}_n(\acute{\phi}), \grave{\phi}), & \text{if } i \geq n \end{cases} \\
 & & \\
c_n(\mathsf{C}_n(\acute{\chi}), \grave{\chi}) & \Rightarrow & \begin{cases} c_i(\mathsf{C}_i(\acute{\chi}), \grave{\chi}), & \text{if } i \leq n \\ c_i(\mathsf{C}_n(\acute{\chi}), \grave{\chi}), & \text{if } i \geq n \end{cases}
\end{array}
$$

The existence of the predicates e, f and c must now be proved. We again use a predicate Φ to express the condition that these predicates are equivalent to the sequences of partial predicates.

Definition 4.5

$$
\begin{array}{l}
\Phi(P, Q, R) \\
\Leftrightarrow \quad \forall \hat{\varepsilon} \,.\, P(\hat{\varepsilon}) \Leftrightarrow \\
\qquad \begin{array}{lcl}
(\grave{\varepsilon} \equiv \bot) & \longrightarrow & \acute{\varepsilon} \equiv \bot, \\
(\grave{\varepsilon} \equiv \underline{?}) & \longrightarrow & \acute{\varepsilon} \sqsubseteq \underline{?}, \\
(\mathit{IsBasic}(\grave{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{B}) \wedge \acute{\varepsilon} \sqsubseteq E(\grave{\varepsilon}), \\
(\mathit{IsFunction}(\grave{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \mathbf{F}) \wedge Q((\acute{\varepsilon} \mid \mathbf{F}), \grave{\varepsilon}), \\
(\mathit{IsCons}(\grave{\varepsilon})) & \longrightarrow & (\acute{\varepsilon} \in \acute{\mathbf{C}}) \wedge R((\acute{\varepsilon} \mid \acute{\mathbf{C}}), \grave{\varepsilon}), \\
\mathit{false} & &
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\wedge\ \forall\hat{\phi}.Q(\hat{\phi}) \Leftrightarrow \\
\quad \bigwedge\{P(\acute{\varepsilon}(\acute{\alpha}),\ \mathit{Eval}(\grave{\varepsilon}_\alpha)) \mid \mathit{Applied}(\grave{\varepsilon}_\alpha,\ \grave{\varepsilon},\ \grave{\alpha}) \\
\qquad\qquad \wedge\ P(\acute{\alpha},\ \mathit{Eval}(\grave{\alpha}))\} \\
\wedge\ \forall\hat{\chi}.R(\hat{\chi}) \Leftrightarrow \\
\quad (\acute{\chi} \equiv \bot) \vee (P(\acute{\varepsilon}_0,\ \mathit{Eval}(\grave{\varepsilon}_1)) \wedge P(\acute{\varepsilon}_1,\ \mathit{Eval}(\grave{\varepsilon}_1))) \\
\qquad \text{where}\ \ ((\gamma, \ell), \delta) = \grave{\chi} \\
\qquad\qquad\quad (\acute{\varepsilon}_0, \acute{\varepsilon}_1) = \acute{\chi} \\
\qquad\qquad\quad \grave{\varepsilon}_i = ((\gamma, \ell_i), \delta) \\
\qquad\qquad\quad (\ell_0, \ell_1) = \gamma(\mathit{Elide}(\gamma, \ell)) \mid \grave{\mathbf{C}}
\end{array}
$$

We may now prove the existence of the required predicates by substituting the predicate sequences into Φ. This then demonstrates that the predicates exist by construction.

Theorem 4.6

$$
\begin{array}{l}
\Phi\ (\lambda\hat{\varepsilon}.\bigwedge_{n=0}^{\infty} e_n(\mathsf{E}_n(\acute{\varepsilon}), \grave{\varepsilon}), \\
\quad \lambda\hat{\phi}.\bigwedge_{n=0}^{\infty} f_n(\mathsf{F}_n(\acute{\phi}), \grave{\phi}), \\
\quad \lambda\hat{\chi}.\bigwedge_{n=0}^{\infty} c_n(\mathsf{C}_n(\acute{\chi}), \grave{\chi}))
\end{array}
$$

is true.

Proof of Theorem 4.6

We demonstrated in the previous chapter that we could substitute the predicate sequences for P and Q.

As an aside we should really prove that the new definition of e_n still satisfies Φ when we are dealing with constructors. This is trivially demonstrated as we are able to distribute $\bigwedge_{n=0}^{\infty}$ through $\wedge$.

Although most of the proof follows from Theorem 3.17, we still have to verify

$$
\begin{aligned}
R(\dot{\chi}) &= \bigwedge_{n=0}^{\infty} c_n(\mathsf{C}_n(\dot{\chi}), \dot{\chi}) \\
&= \bigwedge_{n=0}^{\infty}\{(\dot{\chi} \equiv \bot) \vee\ e_n(\mathsf{E}_n(\dot{\varepsilon}_0), \mathit{Eval}(\dot{\varepsilon}_0)) \\
&\qquad\qquad\qquad\qquad e_n(\mathsf{E}_n(\dot{\varepsilon}_1), \mathit{Eval}(\dot{\varepsilon}_1))\} \\
&= (\dot{\chi} \equiv \bot) \vee\ \bigwedge_{n=0}^{\infty} e_n(\mathsf{E}_n(\dot{\varepsilon}_0), \mathit{Eval}(\dot{\varepsilon}_0)) \wedge \\
&\qquad\qquad\qquad\qquad \bigwedge_{n=0}^{\infty} e_n(\mathsf{E}_n(\dot{\varepsilon}_1), \mathit{Eval}(\dot{\varepsilon}_1)) \\
&= (\dot{\pi} \equiv \bot) \vee P(\dot{\varepsilon}_0, \mathit{Eval}(\dot{\varepsilon}_0)) \wedge P(\dot{\varepsilon}_1, \mathit{Eval}(\dot{\varepsilon}_1)) \\
&\qquad \text{where } ((\gamma, \ell), \delta) = \dot{\chi} \\
&\qquad\qquad\qquad (\dot{\varepsilon}_0, \dot{\varepsilon}_1) = \dot{\chi} \\
&\qquad\qquad\qquad \dot{\varepsilon}_i = ((\gamma, \ell_i), \delta) \\
&\qquad\qquad\qquad (\ell_0, \ell_1) = \gamma(\mathit{Elide}(\gamma, \ell)) \mid \dot{\mathsf{C}}
\end{aligned}
$$

□

This completes the proof that the predicates exist. We may confirm that these predicates do indeed imply the approximation condition by re-proving Corollary 3.20. Again, we are unable to compare functions because of the full abstraction problem with domains associated with the denotational semantics.

The structural induction required to prove the partial congruence that the denotational semantics approximates the operational semantics has been completed in Chapter 3 as Theorem 3.12. This is sufficient because the syntax of the language has not changed, only the initial environment.

4.4 Related Work

The schemes to translate more complicated languages to the one presented here are described in detail by Peyton Jones in [1987]. Hughes' supercombinator abstraction algorithm was first described in [1982a] and further refined in [1983]. Johnsson's lambda-lifter was originally described in [1985] and the translation of pattern-matching for LML is described in [Augustsson 85]. The dependency analysis required for type-checking is outlined in [Peyton Jones 87, pages 118–121]. Polymorphic type checking of the form we have assumed here is described by Hancock in [Peyton Jones 87, Chapters 8–9].

The previous work related to the congruence proof and denotational semantic language definition was identified in the last chapter.

4.5 Conclusion

This concludes our survey of the denotational semantic definition of functional programming langnages; operational realizations of these languages; and the relationship between them. We reiterate that the result we have obtained is the complete congruence between the denotational and operational semantic models we have formally described.

In Part 2, we investigate the practical applications of the extended Theorem 3.3. In Chapter 5 we show that the evaluation order may be changed, in suitable circumstances, without changing the result of the program. In Chapter 6, we consider alternative operational semantics and prove that these machines preserve sharing. Finally we prove that the G-machine compiler for our language is correct in Chapters 7 and 8.

Part II

Applications

Chapter 5

Using Strictness Information

Much work on efficient compilation of lazy functional programming languages has focused on analysis of programs to determine which functions are strict. Various analyses have been proposed to determine such information, which is usually referred to as *strictness information*. In this chapter we discuss the problems associated with using strictness information in our language.

Abstract interpretations are generally defined in terms of the λ-calculus. In the λ-calculus the Second Church-Rosser Theorem [Stoy 77, page 68] provides the relationship between the denotational and operational properties, that is required in abstract interpretation work. The languages to which they are most often applied is that of combinators, with constants and δ-rules. The evaluation mechanism is generally reduction to weak head normal form [Peyton Jones 87, page 198]. This is exactly the language and operational semantics we proved congruent in Chapter 4, and so we may extend the results of abstract interpretations to this language from the more normal λ-calculus with constants.

Typically, an abstract interpretation for strictness will determine which arguments of a user defined function are definitely required to be evaluated. There are a number of gains that may be achieved through using the information provided by strictness analysis. Firstly we are able to improve the space complexity of some algorithms. Consider the function *length* defined with an accumulation parameter.

$$\begin{array}{lll} \mathit{length} & a & [] \quad = a \\ \mathit{length} & a & (x : l) \quad = \mathit{length}\ (a + 1)\ l \end{array}$$

In Lisp the advantage of this definition over the simple definition is that

it runs in constant space. Unless we know that *length* is strict in both parameters the lazy version will run in $\mathcal{O}(n)$ space, as the additions are *not* performed until the end of the list is reached.

The second improvement occurs because we need not create objects on the heap that will immediately be reduced. In functional language implementations heap allocation, with the consequent garbage collection, is often considerably more expensive than stack or register allocation. Any attempt to allocate storage from a stack is therefore likely to be of benefit.

The gain occurs when we have

$$f\,x = g\,((x+3)\times 2)$$

and g is strict. Then instead of constructing the graph associated with $((x+3)\times 2)$ we may go ahead and evaluate it and then proceed with the reduction of g.

5.1 Re-ordering Evaluations in Sequential Machines

We begin with a definition of strictness.

Definition 5.1

A function f is *strict* whenever

$$f\,\bot = \bot$$

It is normal in abstract interpretation work to use complete partially ordered sets, rather than complete lattices, to model domains and to treat the error element as bottom. In the lattice theoretic domain construction we define *complete strictness* in the following way, after [Stoy 77, page 178].

Definition 5.2

A function f is *completely strict* whenever

$$f\,x = \begin{cases} \bot & \text{if } x = \bot \\ \stackrel{?}{=} & \text{if } x = \stackrel{?}{=} \\ \top & \text{if } x = \top \\ f\,x & \text{otherwise} \end{cases}$$

We first define some notation with which to represent the problem.

Definition 5.3

We define $\acute{e} \approx \grave{e}$ if, and only if

$$\acute{e} \sqsupseteq E(\grave{e}) \text{ and } e(\acute{e}).$$

A result from Part I, is that

$$E(\grave{e}) \approx \mathit{Eval}(\grave{e});$$

this follows from Theorems 3.4 and 3.12. From Lemma 3.22, we also have that

$$\acute{e}_i \approx \grave{e}_i \wedge \mathit{Applied}(\grave{e}, \grave{e}_0, \grave{e}_1) \Rightarrow \acute{e}_0\, \acute{e}_1 \approx \grave{e}.$$

The seqnential reordering theorem, Theorem 5.4, says that when a completely strict function is applied to an argument it is permissible to evaluate that argument before evaluating the function body. On a parallel machine we may wish to evaluate the argument in parallel, and this would not change the meaning of the program.

Theorem 5.4

Suppose we have some state $\sigma = ((\gamma, \ell), \delta)$ *with*

$$\gamma(\mathit{Elide}(\gamma, \ell)) \mid \mathbf{A} = (\ell_0, \ell_1).$$

Let $\acute{e}_i = E((\gamma, \ell_i), \delta)$, *and* $\acute{e} = E(\sigma)$. *If* $E((\gamma, \ell_0), \delta)$ *is completely strict, then*

$$E(\mathit{Eval}(\sigma)) \approx \mathit{Eval}((\gamma', \ell), \delta),$$

where $\gamma' = \mathit{fst}(\mathit{fst}(\mathit{Eval}((\gamma, \ell_1), \delta)))$.

We may prove Theorem 5.4 by considering cases for the graph γ' after the evaluation of the argument corresponding to ℓ_1.

Proof of Theorem 5.4

By cases on the value of γ':

$\gamma' = \perp$ **or** $\gamma' \stackrel{?}{=}$ In this case $Eval\,(\mathcal{E}_1) = \gamma'$, hence both $\mathcal{E}'$ and $\mathcal{E}_1$ are also γ'. But $\mathcal{E}_0$ is completely strict, so $\mathcal{E}$ is γ', thus

$$\mathcal{E} \approx \mathcal{E}'.$$

otherwise In this case, $\mathcal{E}_0 \approx ((\gamma', \ell_0), \delta)$ and $\mathcal{E}_1 \approx ((\gamma', \ell_1), \delta)$. From this we may conclude $\mathcal{E} \approx Eval\,((\gamma', \ell), \delta)$.

□

Because of the full abstraction problem it might appear that we have produced a result that is insufficiently powerful; this is not the case. Provided that the program result is not a function, we have shown that either of the operational techniques for reduction will give the same answer.

The result we have proved in Theorem 5.4 is a *dynamic strictness* result. This is because we need to determine the strictness of $E((\gamma, \ell_0), \delta)$ at run-time. This is not in general possible, as we would need to solve the halting problem. Instead we determine a conservative approximation of the strictness, statically, during compilation. Then if $\mathcal{E}\,[\![\mathrm{E}_0]\!]\rho$ is strict we may produce suitable code for $\mathcal{E}\,[\![\mathrm{E}_0\,\mathrm{E}_1]\!]\rho$.

It is important to note that Theorem 5.4 is defined in terms of complete strictness. Strictness abstract interpretations determine only a conservative approximation to the strictness of Definition 5.1. This mis-match in the types of strictness used and provided causes problems when we consider the "program":

$$\Omega\,(1/0) \text{ where } \Omega\,x = \Omega\,x$$

In this case Ω is strict (although not completely strict), but we may not reorder the evaluations, as this will result in $\stackrel{?}{=}$ instead of $\perp$. We note that this occurs because our language no longer has confluence, *i.e.* the Church-Rosser property.

5.2 Related Work

The use of abstract interpretation to deduce approximations to fixed points in lattices was first suggested by the Cousots in [1977]. This general work on lattice fixed points was applied to functional programs by Mycroft in [1981a] and [1981b]. This work was fairly limited, to the extent that the approximate result it produced was fairly weak. Subsequent work has devoted attention to improving on the quality and range of the information derived. Extending

the analysis to cover higher-order functions is described in [Burn *et al.* 86]. Further refinements to cover data structures are described in [Wadler 87]. All of these works are justified because of an appeal to the Church-Rosser Theorems for the λ-calculus. This provides the relationship between operational and denotational semantic models. This is needed as the analysis occurs by variation of the denotational semantics value domains and the use of this information occurs in the operational semantics.

5.3 Conclusion

The most important observation is that the language with which we are working is not confluent, *i.e.* it does not have the Church-Rosser Property. This occurs because there are circumstances where both bottom and error could be returned, depending on which reduction order is chosen. Confluence is lost when bottom is returned for error.

There are two simple ways that we may recover the property of confluence. The first is to remove the distinction between error and bottom, so that we do not report errors at all. This has the advantage that the underlying λ-calculus model behaves in a similar manner. Using the normal definition of division in the λ-calculus means that division by zero is represented by non-termination. The second solution is to introduce a new node type to represent error. In this case an error is slowly propagated up to become the final result. This is precisely how erroneous values in the denotational semantics are handled, so we should not be too surprised by this solution. This has an advantage when the propagation process is blocked by non-termination, thus providing the correct result. As an example we can consider the reordered evaluation of $\lambda x.\bot \stackrel{?}{=}$ mentioned earlier in this chapter, which under the proposed new scheme produces the correct result. The main drawback of the second solution is that we are generally unable to represent error in the V stack, and so correct error handling by the optimizations of Chapter 8 becomes quite complicated.

Even though the language is non-confluent, we may reorder the evaluation of any completely strict function, as a result of Theorem 5.4. This causes a problem as most abstract interpretation work does not derive complete strictness information. Again the solutions proposed above will retrieve the situation.

Chapter 6

Sharing Mechanisms

Sharing is that property of graph reduction that makes the implementation of lazy functional programming languages practicable. Without sharing, we must recompute the value of an argument each time it is used, because we are performing call-by-name evaluation. Such a reduction strategy is unacceptably slow on a sequential machine, although it may be adequate for a parallel one, see [Downey and Sethi 76].

The simplest example of sharing occurs in the following program:

$$f\,(1+2)$$
$$\text{where } f\,x = x + x.$$

If the function f does not share its argument x, then we will need to evaluate the expression $(1+2)$ twice. Sharing is accomplished in the implementation we defined in Chapter 2 by using a graph to represent the state of the computation. In the case we are currently considering, a single node will represent $(1+2)$ and upon its reduction the result 3 will be used to overwrite the node. Any references to the expression $(1+2)$ will be *via* this node, and thus all such references will benefit from the reduction of $(1+2)$ to 3.

We call this property *sharing*. Any reduction that is performed without the result overwriting the root node may cause sharing to be lost. We note that as the operational semantics is currently defined in Chapter 2, or the modified version in Chapter 4, this can never happen; the reason is that *all* reduction steps write out the result to the root node at the end of the *Step* function. As we shall see in the chapters on the G-machine, there are optimization techniques which will be available whenever we are allowed

to omit the updating of root nodes. This is also the key observation in the Spineless G-machine, where optimizations are proposed to reduce heap accesses by omitting the unnecessary updating of nodes. In the spineless G-machine updating occurs only when sharing is possible.

6.1 A Spine Cycle Theorem

In this section we discuss a result which relates the form of the spine for a state and the meaning of the state when derepresented by E. We wish to show that a node can appear only once in the dump, if the program is to terminate. Informally this arises because a node can only appear on the dump when we are attempting to reduce it to WHNF. Thus when a node appears twice, its reduction requires that it already be in WHNF. This process can not terminate.

This result is required when we wish to show that the whole of the dump need not be reconstructed for each step. Without this result, the updating associated with reduction steps might affect the way that the dump was constructed. Another application, which we use in this chapter, is to show that we may postpone the updating until the body of the combinator has reached WHNF.

We first define the function *Redex* which calculates the root of the indirection node that will be created at the end of the next step.

Definition 6.1

$$
\begin{array}{rcll}
\mathit{Redex}(\sigma) & = & (\mathrm{I} = [\![\mathbf{null}]\!]) & \longrightarrow (\ell : \phi) \,!\, 1, \\
& & (\mathrm{I} = [\![\mathbf{head}]\!]) & \longrightarrow (\ell : \phi) \,!\, 1, \\
& & (\mathrm{I} = [\![\mathbf{cons}]\!]) & \longrightarrow (\ell : \phi) \,!\, 2, \\
& & (\mathrm{I} = [\![+]\!]) & \longrightarrow (\ell : \phi) \,!\, 2, \\
& & (\mathrm{I} = [\![=]\!]) & \longrightarrow (\ell : \phi) \,!\, 2, \\
& & (\mathrm{I} = [\![\mathbf{if}]\!]) & \longrightarrow (\ell : \phi) \,!\, 3, \\
& & & \quad (\ell : \phi) \,!\, (\mathit{Args}(\delta\, [\![\mathrm{I}]\!])) \\
& \text{where} & (\ell : \phi) : \psi & = \mathit{Dump}(\sigma) \\
& & \mathrm{I} & = \gamma\ell \mid \mathrm{Ide} \\
& & ((\gamma, r), \delta) & = \sigma
\end{array}
$$

The result we shall prove is that the spine of a state σ will have no cycles whenever $E(\sigma)$ is not $\perp$. This arises because if σ has spine cycles, $\mathit{Step}(\sigma) = \perp$. This is formally stated as Theorem 6.2.

Theorem 6.2

Let

$$\sigma_{n+1} = Done(\sigma_n) \longrightarrow \sigma_n,\ Step(\sigma_n) = ((\gamma_{n+1}, r), \delta),$$

$\psi_n = Dump(\sigma_n)$ *and* $o_n = Redex(\sigma_n)$. *Then if* o_n *is overwritten with an indirection node to* r, $\sigma_{n+1} = \bot$.

We note that as our extended machine is defined, there are no cycles in the graph. This automatically ensures that there are no cycles in the spines. If cycles in graphs are permitted we would be able to appeal to the typechecker to eliminate all cycles except the trivial ones corresponding to the programs like:

$$\begin{array}{lllll} x & \text{whererec} & x & = & x \\ x & \text{whererec} & x & = & x+1 \end{array}$$

Although we have restricted the range of programs which exhibit cycles in the spine, we have not eliminated them from consideration. For this reason we prove Theorem 6.2 in all its generality, by showing that $dump(\sigma_{n+1}) = \bot$ whenever σ_n has a spine cycle. We notice that the redex will be an initial sequence of the first element of the dump. Because the dump is constructed from the root to the redex, we must have both that $\#\,\psi_{n+1} \neq \bot$ and that $\#\,(head\,\psi_{n+1}) \neq \bot$ in order for $Dump\,(\sigma_{n+1})$ to terminate.

Proof of Theorem 6.2

We consider four cases: whether the dump is a singleton list or not; and whether the redex is the last element of the current stack or not. We note that only cases 1 and 3 are permissible in a typechecked implementation.

1. $\underline{\psi_n = [\phi'' +\!\!+ [o_n]]}$ In this case $o_n = r$ and we have $\psi_{n+1} = [\bot]$. Thus $\#\,(head\,\psi_{n+1}) = \bot$, which implies that we have $\sigma_{n+1} = \bot$.
2. $\underline{\psi_n = [\phi'' +\!\!+ [o_n] +\!\!+ \phi' +\!\!+ [r]]}$ In this case

   $$\psi_{n+1} = [head\,\psi_{n+1} +\!\!+ \phi' +\!\!+ [r]]$$

   and so again $\#\,(head\,\psi_{n+1}) = \bot$, which implies that we have $\sigma_{n+1} = \bot$.

3. $\underline{\psi_n = [\phi'' +\!\!+ [o_n]] +\!\!+ \psi' +\!\!+ [\phi''' +\!\!+ [r]]}$ In this case

$$\psi_{n+1} = \psi_{n+1} +\!\!+ \psi' +\!\!+ [\phi''' +\!\!+ [r]]$$

This time $\# \psi_{n+1} = \perp$; the result is that $\sigma_{n+1} = \perp$.

4. $\underline{\psi_n = [\phi'' +\!\!+ [o_n] +\!\!+ \phi'] +\!\!+ \psi' +\!\!+ [\phi''' +\!\!+ [r]]}$ In this case

$$\psi_{n+1} = \psi'' +\!\!+ \psi' +\!\!+ [\phi''' +\!\!+ [r]]$$

where $\psi'' = \psi'' +\!\!+ \psi' +\!\!+ [\phi''' +\!\!+ [r] +\!\!+ \phi']$, which again implies $\sigma_{n+1} = \perp$, because $\# \psi_{n+1} = \perp$.

□

We conclude this section by restating the result. In a terminating program there are no cycles in the spine or dump for any initial, intermediate or final state representing the execution of that program.

6.2 A Weak Head Normal Form Theorem

We now wish to demonstrate that the operational semantics has the following property: any node on the dump of a state may be evaluated before the remainder of the evaluation is performed, and the result will be the same. Informally this is because every node on a dump is reduced to weak head normal form before it is released from the dump. First we define a function *EvalFrom* which evaluates from the node provided in its first argument, using the state supplied by the second.

Definition 6.3

$$\begin{aligned} \mathit{EvalFrom}\,\ell\,((\gamma, r), \delta) = \; & ((\gamma', r), \delta) \\ & \text{where } ((\gamma', \ell), \delta) = \mathit{Eval}((\gamma, \ell), \delta) \end{aligned}$$

We may now formally state our theorem as Theorem 6.4.

Theorem 6.4

Suppose that ℓ occurs in $Dump(\sigma)$. Then

$$\mathit{Eval}(\sigma) = \mathit{Eval}(\mathit{EvalFrom}\,\ell\,\sigma)$$

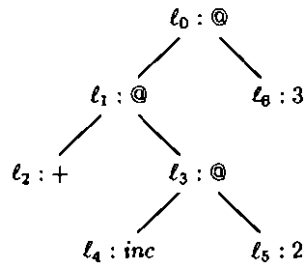

Figure 6.1: Example of a Graph

Consider the example of the graph in Figure 6.1. This has as a dump $[[\ell_4, \ell_3], [\ell_2, \ell_1, \ell_0]]$. If we attempt to reduce any of these nodes to WHNF before proceeding with the evaluation of the whole graph, our theorem says that we will get the same resulting state as we would by evaluating the whole graph in the normal way. We prove this result by showing that there is a stepwise correspondence between the execution of the two functions on a given state.

Proof of Theorem 6.4

We consider two cases of $EvalFrom\,\ell\,\sigma$:

$= \perp$ Let

$$H = \lambda\kappa\lambda\sigma.Done(\sigma) \longrightarrow \sigma, \kappa(Step(\sigma)),$$

and $\sigma_\ell = ((\gamma, \ell), \delta)$. Then, either

$\nexists n$ **with** $H^n \perp \sigma_\ell = H^{n+1} \perp \sigma_\ell$ In this case $\nexists n$ **with** $H^n \perp \sigma = H^{n+1} \perp \sigma$.

$\exists n$ **with** $H^{n+1} \perp \sigma_\ell = \perp$ This occurs because $Dump(H^n \perp \sigma_\ell) = \perp$. But this means that $Dump(H^n \perp \sigma) = \perp$ also.

$\neq \perp$ In this case execution of *EvalFrom* must terminate after some finite number of steps. Let us denote this number by n, and define H as

$$H = \lambda\kappa\lambda\sigma.Done(\sigma) \longrightarrow \sigma, \kappa(Step(\sigma))$$

Then $EvalFrom\,\ell((\gamma, r), \delta) = ((\gamma', r), \delta)$ where

$$((\gamma', \ell), \delta) = H^{n+1}(\lambda\sigma.\sigma)((\gamma, \ell), \delta)$$

But $Eval(\sigma) = H^{n+1}\,Eval\,\sigma$, and we must now show that the individual steps associated with each H are the same. This follows from the observation that $Redex(\sigma_\ell) = Redex(\sigma)$ and that the graphs are the same. Observe that we are using the Spine Cycle Theorem (Theorem 6.2) to establish this.

□

This is an important result. We shall use it to allow us to reorder evaluations in this chapter as well as Chapter 7.

6.3 Copying Shared Nodes

We now wish to demonstrate that we may use an alternative algorithm to maintain the sharing information in the state. This algorithm first evaluates the body of a function definition and then updates the original root of the redex with the correct value. The potential problem with this approach is that there is a "hole" in the graph γ until evaluation is completed. This is because we have not updated the value of the root for all the intermediate steps. We will fall into the hole in the graph only when there is a cycle in the spine. From Theorem 6.2 we know that the result of the evaluation in this case will be $\perp$. We would like to show that our reordered evaluation will also produce the same value. It does; although it converts call-by-need into call-by-name, when there is a cycle. Informally, we may say that the original operational semantics evaluates a state with a spine cycle more rapidly than the new operational semantics with copying. Formally we have Theorem 6.5.

Theorem 6.5

Let $\sigma_i = ((\gamma_i, r), \delta)$ with $Done(\sigma_0)$ false, and $\sigma_{i+1} = Step(\sigma_i)$. Then we define $\nu_i = \gamma_i o_0$ where $o_0 = Redex(\sigma_0)$. Let $\nu_1 = \ell$ in N, be the indirection node associated with the reduction, and define $\gamma_1' = \gamma_1 \oplus \{o_0 \mapsto \nu_0\}$, so that γ_1' differs from γ_1 only in that the indirection associated with the reduction has not been performed. Then

$$Eval(\sigma_0) = Eval((\gamma' \oplus \{o_0 \mapsto \nu_1\}, r), \delta),$$

where $((\gamma', o_0), \delta) = Eval((\gamma_1', o_0), \delta)$.

The proof is straightforward. As the two graphs differ only in the value they assign to o_0, we use fixpoint induction on the uses of *Eval* to show that this is unimportant unless there is a cycle – in which case it remains unimportant because we have non-termination.

Proof of Theorem 6.5

We first observe that

$$Eval(\sigma_0) = Eval((\gamma, r), \delta)$$

where $((\gamma, o_0), \delta) = Eval((\gamma_1, o_0), \delta)$. But

$$Eval((\gamma_1', o_0), \delta) = Eval((\gamma_1, o_0), \delta),$$

by fixpoint induction and using the spine cycle theorem, Theorem 6.2.

□

This does not provide a realistic implementation method, although it does perform the correct function. If, instead of creating an indirection node on the completion of the evaluation, we make a copy of the result over the original root node, we have another implementation. Although this loses sharing, it is not *significant* sharing. Sharing is significant if its loss results in more reductions being performed. By this measure the copying of a node that has been evaluated to WHNF is non-significant. This is because no further reductions of it are possible and thus fresh attempts to reduce either the copy or the original will cause no further reduction to be performed. We have therefore justified the copying implementation of graph reduction, which is used in SKIM [Stoye *et al.* 84] and the G-machine [Johnsson 83, Johnsson 87][1].

[1] A weaker form of the graph-isomorphism proposed in Chapter 8 can be devised. It would allow us to "common up" separate occurrences of WHNF nodes which were graph-isomorphic. As this property only occurs in the current discussion, and would have no further practical applications, it has been omitted.

Although our original operational semantics is defined using indirection nodes throughout, we would not suggest this method as a practical real implementation method. The main modification that should be made is to copy base values, and use indirection nodes only when the returned value may be a function. The result of this section (Theorem 6.5) justifies the use of this composite scheme.

This modified copying and indirection updating scheme is still suitable for implementing Hughes' lazy memo-functions [1985].

6.4 Indirections Without Chaining

It is difficult to prove anything concerning the relative efficiencies of the copying update and the composite indirection and copying schemes. The reason is that the relative merits of the schemes depends on the style of programming used. If functions are predominantly used with base values then the copying scheme is better. If heavy use is made of higher order functions, then the indirection and copying scheme will work best.

The problem for the purely copying updating scheme, occurs when a functional valued result is obtained. In this case we have the overhead of stack frame creation and deletion, when compared with the indirection node updating method. For the indirection node method there is the well known problem associated with the accumulation of indirection node chains. Here access times to function arguments may be slowed, because we must search along a chain of indirection nodes to find the argument.

In an attempt to avoid chains of indirection nodes occurring, we outline the following scheme. We also avoid the stack overheads associated with the body reduction discussed in the previous section. There are two principal observations.

1. We may invert the direction that indirection nodes point. We are, after all, using indirection nodes to equivalence two nodes in the graph. So the direction of the indirection node is unimportant.

2. Only two sorts of indirection nodes exist: those pointing to nodes that are in WHNF, and those pointing to nodes that are being evaluated, but have not reached WHNF yet.

Imagine, then, that we have reached a stage in the computation in which we are about to replace a redex with its result and consider the three possible states for the body:

WHNF We may copy the root of the body to the root of the redex.

non-WHNF, unevaluated Because no attempt has been made to reduce it, this node has no indirection node pointing to it.

non-WHNF being-evaluated In this case the state will eventually become $\perp$. We will ignore the creation of chains of indirection nodes in non-terminating programs.

This suggests a different definition of the *Update* function.

$$\begin{aligned} &\mathit{Update}\,\ell\,\ell'\,((\gamma, \tau), \delta) \\ &\quad = \mathit{Done}((\gamma, \ell'), \delta) \quad \longrightarrow \quad ((\gamma \oplus \{\ell \mapsto \gamma\,\ell'\}, \tau), \delta), \\ &\qquad\qquad\qquad\qquad\qquad\quad ((\gamma \oplus \{\ell \mapsto \gamma\,\ell', \ell' \mapsto \ell\ \mathbf{in}\ \mathrm{N}\}, \tau), \delta) \end{aligned}$$

That is, an indirection node is only inserted when further reduction of the body can occur. Of course the overwriting of ℓ or ℓ' or both may sometimes be omitted, but this will depend on the results of a sharing analysis.

The important points about this technique are:

1. Chains of indirection nodes do not accumulate during the execution of terminating programs, although they may do so for some non-terminating ones.

2. Higher order functions are dealt with in a completely natural way. In the G-machine stack operations cause a significant overhead for the correct sharing of higher order functions.

The equivalence of this scheme to the original indirection node model is guaranteed because both alternatives in the conditional have been shown to provide the same result. The operational aspects of the new scheme have been justified by appealing to the various situations in which indirection nodes can arise.

6.5 Related Work

The SKIM machine [Clarke *et al.* 80] was the first description of a machine in which reduction of the body of a function was proposed as a solution to the problem of sharing in projection functions [Stoye *et al.* 84]. The problem for SKI reduction is particularly acute as most reductions are of projection

functions, and the granularity of reductions is a lot finer than those described in this thesis. It was taken up independently by Augustsson and Johnsson in their G-machine [Johnsson 83] and [Johnsson 84]. The equivalence of this technique to the tail recursive scheme described in this thesis is proved in a G-machine setting in [Lester 85] and [Lester 87].

Analysing programs to detect sharing is described in [Goldberg 87] where an abstract interpretation is used to find a superset of the nodes that will be shared during reduction. Burn, Peyton Jones and Robson describe the use of such information within a G-machine setting in [1988]; they call the resulting machine a Spineless G-machine.

6.6 Conclusion

The final proposal for reduction was inspired by Hancock's observation that higher order function reduction in a G-machine can be slow and that the judicious use of indirection nodes might reduce the overheads. The fact that we have no chaining of indirection nodes means that this is the method of choice for higher order functions, although copying will work best for base type results.

It appears that the the use of indirection nodes will be of less value within a Spineless G-machine, although further work is required to provide a definitive answer.

Chapter 7

Deriving the G-Machine

In this chapter we show how the G-machine [Johnsson 83] may be derived from the operational semantics that we defined in Chapter 4. Viewed as an implementation method this operational semantics has disadvantages. Initially the main problem is that it calculates the next redex in its reduction strategy by starting at the root of the graph. This is unnecessary if we keep a stack of separate frames to record the current depth in the recursive calls to the interpreter. The equivalence of these two definitions is established in Section 7.1. The new reduction strategy is obviously more efficient, but it may be further refined by observing that part of the stack remains unchanged during a reduction step. This is the part that is above the root of the redex – a result that is another application of the Spine Cycle Theorem. This improvement is derived in Section 7.2.

In Section 7.3 we show how individual reduction steps may be performed using compiled code rather than using an interpretive technique, of Chapter 3. In the interpreter a syntactic representation of the combinator is included in the environment. The details are in Chapter 3, and the method is essentially similar to template instantiation which is described in [Peyton Jones 87].

Finally we show that the G-machine *print* instruction may be added. This is important if the program result may be a structured data object, as we need to provide a driver to initiate demand in the program.

7.1 Another Interpreter

Before we derive the G-machine we re-cast the definition of the operational semantics. The definition we have been using so far has the pleasing property that we can restrict the number of reductions performed easily. This leads to clean proofs by fixpoint induction, which we have used extensively. We would now like to transform this definition to a form more suitable for a real implementation. The change that is introduced in this section is to define *Step* and the associated built-in step functions $*Step$ in terms of the spine rather than the dump. To do this we must evaluate strict arguments to built-in functions recursively. It is this explicit recursion which complicates fixpoint induction proofs were we to use such a definition. There is, after all, no limit to the depth of such recursion unless we invoke the Spine Cycle Theorem. The new *Step* and $*Step$ functions are defined in Figures 7.1 and 7.2.

We must now show the congruence of these definitions; this is done in Theorem 7.1.

Theorem 7.1

$$Eval' = Eval$$
$$\text{where } Eval'(\sigma) = Done(\sigma) \longrightarrow \sigma,\ Eval(Step'(\sigma)).$$

We are unable to show the stepwise equivalence of this new definition with the old, because $Step'$ may perform a considerable number of reductions in reducing a built-in function. It is clear that *Step* and $Step'$ differ only in their treatment of strict built-in functions. We therefore consider the five special cases that arise.

Proof of Theorem 7.1

We first observe that: $Eval'(\sigma) = Eval(\sigma)$ if and only if

$$Eval(Step'(\sigma)) = Eval(\sigma).$$

Clearly, two cases are trivial:

$$Step'(\sigma) = \left\{ \begin{array}{l} ConsStep\ \phi\ \sigma \\ OtherStep\ (\delta\ [\![\mathrm{I}]\!])\ \phi\ \sigma \end{array} \right\} = Step(\sigma).$$

$$
\begin{array}{lll}
Step'(\sigma) = & (\mathrm{I} = [\![\mathrm{null}]\!]) & \longrightarrow \; NullStep'\,\phi\,\sigma, \\
& (\mathrm{I} = [\![\mathrm{head}]\!]) & \longrightarrow \; HeadStep'\,\phi\,\sigma, \\
& (\mathrm{I} = [\![\mathrm{cons}]\!]) & \longrightarrow \; ConsStep\,\phi\,\sigma, \\
& (\mathrm{I} = [\![+]\!]) & \longrightarrow \; PlusStep'\,\phi\,\sigma, \\
& (\mathrm{I} = [\![=]\!]) & \longrightarrow \; EqStep'\,\phi\,\sigma, \\
& (\mathrm{I} = [\![\mathrm{if}]\!]) & \longrightarrow \; IfStep'\,\phi\,\sigma, \\
& & \quad OtherStep\,(\delta\,[\![\mathrm{I}]\!])\,\phi\,\sigma \\
& \text{where} \quad (\ell : \phi) & = \; Spine(\gamma, r) \\
& ((\gamma, r), \delta) & = \; \sigma \\
& \mathrm{I} & = \; (\gamma\,\ell) \mid \mathrm{Ide}
\end{array}
$$

Figure 7.1: The *Step'* function

$$
\begin{array}{lcl}
NullStep'\,(\ell : \phi) & = & NullStep\,(\ell : \phi) \circ EvalArg\,\ell \\
HeadStep'\,(\ell : \phi) & = & HeadStep\,(\ell : \phi) \circ EvalArg\,\ell \\
PlusStep'\,(\ell_0 : \ell_1 : \phi) & = & PlusStep\,(\ell_0 : \ell_1 : \phi) \circ EvalArg\,\ell_1 \\
 & & \quad \circ\, EvalArg\,\ell_0 \\
EqStep'\,(\ell_0 : \ell_1 : \phi) & = & EqStep\,(\ell_0 : \ell_1 : \phi) \circ EvalArg\,\ell_1 \\
 & & \quad \circ\, EvalArg\,\ell_0 \\
IfStep'\,(\ell_0 : \ell_1 : \ell_2 : \phi) & = & IfStep\,(\ell_0 : \ell_1 : \ell_2 : \phi) \circ EvalArg\,\ell_0 \\
 & & \\
EvalArg\,\ell\,((\gamma, r), \delta) & = & EvalFrom\,(Arg(\gamma, \ell))((\gamma', r), \delta)
\end{array}
$$

Figure 7.2: The Built-in Reductions: $*Step'$

There are five other cases, which we divide into two groups according to the number of strict arguments.

1. Suppose that $Step'(\sigma) = NullStep'\,(\ell : \phi)\,\sigma$. Then

   $$NullStep'\,(\ell : \phi)\,\sigma = EvalFrom\,\ell\,\sigma.$$

   But ℓ occurs in the dump of the state σ, and hence by the WHNF Theorem (6.4), we have: $Eval\,\sigma = Eval\,(EvalFrom\,\ell\,\sigma)$. And thus

   $$Eval'\,(\sigma) = Eval\,(\sigma).$$

   The steps for *HeadStep'* and *IfStep'* are similar.

2. We now consider the built-in functions that are strict in two arguments. In this case we must perform the reduction to weak head normal form twice. Let $\ell_i' = Arg\,(\gamma, \ell_i)$, and assume that

$((\gamma, r), \delta) = \sigma$. Let $\sigma_0 = ((\gamma, \ell_0'), \delta)$ and $\sigma_1 = ((\gamma', \ell_1'), \delta)$ where $((\gamma', \ell_1'), \delta) = \mathit{EvalFrom}\, \ell_0'\, \sigma$.

Suppose that the built-in function is addition. Then Eval' becomes $\mathit{Eval} \circ (\mathit{PlusStep}'\, (\ell_0 : \ell_1 : \phi))$. This may be further expanded to $\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi) \circ \mathit{EvalArg}\, \ell_1 \circ \mathit{EvalArg}\, \ell_0$. These steps follow from the definitions of Eval' and $\mathit{PlusStep}'$.

Again we would like to show that $\mathit{Eval}'\, \sigma = \mathit{Eval}\, \sigma$. This is achieved by appealing to the WHNF Theorem.

We first consider the possible values that $\mathit{Done}\, (\sigma_0)$ may have. If the value is improper then the complete strictness of $\mathit{EvalArg}$ and $\mathit{PlusStep}$ ensures that this improper value is propagated through the composition of functions. We therefore consider the proper values:

(a) $\mathit{Done}\, (\sigma_0)$ is true. This means that the first argument is already in WHNF. Thus Eval' becomes
$\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi) \circ \mathit{EvalArg}\, \ell_1$.

(b) $\mathit{Done}\, (\sigma_0)$ is false. In this case the first argument must be reduced to WHNF. But ℓ_0' is in the dump of σ, and by the WHNF Theorem $\mathit{Eval}\, \sigma = \mathit{Eval}\, \sigma'$ where $\sigma' = \mathit{EvalFrom}\, \ell_0'\, \sigma$.

We must now show that $\mathit{Eval}'\, \sigma'$ is
$(\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi) \circ \mathit{EvalArg}\, \ell_1)\, \sigma'$.

This result follows from a second case analysis; this time on the proper values that $\mathit{Done}\, (\sigma_1)$ may return.

(a) $\mathit{Done}\, (\sigma_1)$ is true. This means that the second argument is already in WHNF. Thus $\mathit{Eval}'\, (\sigma')$ now becomes
$(\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi))\, \sigma'$.

(b) $\mathit{Done}\, (\sigma_1)$ is false. In this case the second argument must be reduced to WHNF. But ℓ_1' is in the dump of σ, and by the WHNF Theorem
$\mathit{Eval}\, \sigma' = \mathit{Eval}\, \sigma''$ where $\sigma'' = \mathit{EvalFrom}\, \ell_1'\, \sigma'$.

As we have now shown that
$\mathit{Eval}'(\sigma) = (\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi))\, \sigma''$
and
$\mathit{Eval}(\sigma) = (\mathit{Eval} \circ \mathit{PlusStep}\, (\ell_0 : \ell_1 : \phi))\, \sigma''$
we are able to conclude that the functions Eval and Eval' are the same, when the built-in function is addition.

The proof of the equivalent result for EqStep is similar.

□

We have now introduced a distinction between reduction steps used to reduce the current stack function, and those used to reduce strict arguments required by the stack function. We now consider ways to improve the unwinding operation. This calculates the root of the next redex and makes the arguments to the function accessible through the stack.

7.2 Fast Unwinding

In this section we show how the unwinding operation may be made more efficient. We currently recompute the spine of a graph for each reduction step; this is unnecessary because most of it will remain constant between steps. In Figure 7.2 the domain of a G-machine state is defined. This includes components with which we are currently not interested, such as the **O** and **V** components. However, by using the more general state, we need not redefine the combinators that represent instructions each time we expand the state to use these components.

In this section we shall be especially concerned with the *unwind* instruction. We would like to show that it implements the house-keeping detail associated with the spine. In Figure 7.4 we give the first definition of *unwind* which we label $\underline{unwind_1}$. Notice that it uses a subsidiary instruction $\underline{unwind'_1}$ to unwind the stack and that the argument check is performed by the *entry* instruction. Also notice that it does not re-arrange the stack to make access to the function's arguments more direct. This contrasts with the *unwind* instruction defined by Johnsson [1983] and Peyton Jones [1987].

Note that we shall use ρ as the variable name for an element of $\mathbf{D}$ for the original operational semantics global environment. Because the states are obtained from two separate domains we define an equivalence condition. We define this equivalence as $\cong$ in Definition 7.2. We recall that the accent convention allows us to refer to similar objects in two different implementations. We will now accent the operational semantics objects with an acute accent and the corresponding G-machine objects with a grave accent.

Definition 7.2

Suppose $\acute{\sigma} = ((\acute{\gamma}, \tau), \acute{\rho})$ and $\grave{\sigma} = (o, \phi, \psi, \grave{\gamma}, \grave{\rho}, \delta)$. Then $\acute{\sigma} \cong \grave{\sigma}$, if and only if, $\acute{\gamma} = \grave{\gamma}$, $\acute{\rho} = \grave{\rho}$ and $\tau = last\,(last\,(\phi : \delta))$.

σ	$\in$ **S**	=	$\mathbf{O} \times \mathbf{L}^* \times \mathbf{V} \times \mathbf{G} \times \mathbf{U} \times \mathbf{D}$	(States)
o	$\in$ **O**	=	$\mathbf{Z}^*$	(Output)
ψ	$\in$ **V**	=	$(\mathbf{Z} + \mathbf{T})^*$	(Value Stack)
γ	$\in$ **G**	=	$[\mathbf{L} \to \mathbf{N}]$	(Graph Maps)
ν	$\in$ **N**	=	$\mathbf{A} + \mathbf{I} + \mathrm{Ide} + \mathbf{Z} + \mathbf{T} + \mathbf{C} + \{nil\}$	(Nodes)
	A	=	$\mathbf{L} \times \mathbf{L}$	(Application Nodes)
	I	=	**L**	(Indirection Nodes)
	Z	=	$\{\ldots, -1, 0, 1, \ldots\}_\perp$	(Integers)
	T	=	$\{true, false\}_\perp$	(Truth Values)
	C	=	$\mathbf{L} \times \mathbf{L}$	(Constructor Nodes)
ρ	$\in$ **U**	=	$[\mathrm{Ide} \to \mathrm{Comb}]$	(Environments)
δ	$\in$ **D**	=	$\mathbf{L}^{**}$	(Dumps)
ϕ	$\in$ $\mathbf{L}^*$			(Stacks)
κ	$\in$ **K**	=	$[\mathbf{S} \to \mathbf{S}]$	(Continuations)
β	$\in$ **B**	=	$[\mathrm{Ide} \to \mathbf{Z}]$	(Bindings)
ℓ	$\in$ **L**			(Node Labels)

Figure 7.3: Value Domains

We now establish Theorem 7.3. Informally this demonstrates that the concept of evaluation in both semantics is an equivalence preserving operation.

Theorem 7.3

If $\grave{\sigma} = (o, \phi, \psi, \gamma, \rho, \delta)$ *and* $\acute{\sigma} \cong \grave{\sigma}$, *then*

$$EvalFrom\,(last\,\phi)\,\acute{\sigma} \cong fix(\underline{unwind}_1)\,\grave{\sigma}$$

Before this is proved we must establish that the *Spine* function and $\underline{unwind}_1$ create the same stack from equivalent states.

Lemma 7.4

Suppose that $\acute{\sigma} = ((\gamma, r), \rho)$, *that* $\grave{\sigma} = (o, \phi, \psi, \gamma, \rho, \delta)$, *and that* $\acute{\sigma} \cong \grave{\sigma}$. *Then* $Spine(\gamma, last\,\phi) = \phi'$, *if and only if*

$$
\begin{array}{lcl}
\underline{entry}\, n\, \kappa\, \sigma & = & \#\phi \geq n \longrightarrow \kappa\sigma, \\
 & & \quad \sigma \\
 & & \text{where } (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{unwind}_1\, \kappa\, \sigma & = & (\nu \in \text{Ide}) \longrightarrow \kappa'\, \sigma', \sigma' \\
 & & \quad \text{where } \kappa' = \underline{entry}\, n (\kappa \circ \underline{pop}\, n \circ \underline{step}) \\
 & & \qquad n = \mathit{ArgsGM}\, (\nu \mid \text{Ide})\, \rho \\
 & & \qquad \nu = \gamma\, \ell \\
 & & \qquad \sigma' = \mathit{fix}(\underline{unwind}'_1)\, \sigma \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma' \\
 & & \\
\underline{unwind}'_1\, \kappa\, \sigma & = & (\nu \in \mathbf{I}) \longrightarrow \kappa (o, (\nu \mid \mathbf{I}) : \phi, \psi, \gamma, \rho, \delta), \\
 & & (\nu \in \mathbf{A}) \longrightarrow \kappa (o, \mathit{fst} (\nu \mid \mathbf{A}) : \phi, \psi, \gamma, \rho, \delta), \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } \nu = \gamma\, \ell \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{pop}\, n\, \sigma & = & (o, \mathit{drop}\, n\, \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } (o, \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{step}\, \sigma & = & (o, \phi, \psi, \gamma', \rho, \delta) \\
 & & \quad \text{where } ((\gamma', r), \rho) = \mathit{Step}' ((\gamma, r), \rho) \\
 & & \qquad r = \mathit{last}\, (\phi) \\
 & & \qquad (o, \phi, \psi, \gamma, \rho, \delta) = \sigma
\end{array}
$$

Figure 7.4: Instructions for Function Call and Return

$$\mathit{fix}(\underline{unwind}'_1)(\dot{\sigma}) = (o, \phi', \psi, \gamma, \rho, \delta).$$

This is a simple example of fixpoint induction. There are two loops in the operational semantics. The outer one performs steps until a Weak Head Normal Form is reached; the inner one performs the spine unwinding operation for each step. Both of these are defined using a fixpoint. In Lemma 7.4, we are showing that the inner loop of the operational model of graph reduction is equivalent in both implementations.

Proof of Lemma 7.4

Simple dual fixpoint induction.

$$
\begin{array}{lcll}
\mathit{ArgsGM}\,\mathrm{I}\,\dot{\rho} & = & (\mathrm{I} = [\![\mathrm{null}]\!]) & \longrightarrow\ 1, \\
 & & (\mathrm{I} = [\![\mathrm{head}]\!]) & \longrightarrow\ 1, \\
 & & (\mathrm{I} = [\![\mathrm{cons}]\!]) & \longrightarrow\ 2, \\
 & & (\mathrm{I} = [\![+]\!]) & \longrightarrow\ 2, \\
 & & (\mathrm{I} = [\![=]\!]) & \longrightarrow\ 2, \\
 & & (\mathrm{I} = [\![\mathrm{if}]\!]) & \longrightarrow\ 3, \\
 & & & (\mathit{Args}(\dot{\rho}\,[\![\mathrm{I}]\!]))
\end{array}
$$

Figure 7.5: The *ArgsGM* function

$$
\begin{array}{lcl}
\underline{\mathit{eval}}\sigma & = & (\underline{\mathit{restore}} \circ \mathit{fix}(\underline{\mathit{unwind}}_1))(o, [\ell], \psi, \gamma, \rho, \phi : \delta) \\
 & & \quad \text{where } (o, \ell : \phi, \psi, \gamma, \rho, \delta) \ = \ \sigma \\
 & & \\
\underline{\mathit{restore}}\,\sigma & = & (o, \mathit{last}\,\phi : \phi', \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } (o, \phi, \psi, \gamma, \rho, \phi' : \delta) = \sigma
\end{array}
$$

Figure 7.6: The *eval* and *restore* Instructions

□

We will now establisb that a sequence of reduction steps in each semantics preserves the equivalence condition $\cong$. Informally it is the outer loop of tbe interpreters that we are now showing to be equivalent. The proof is a fixpoint induction.

Proof of Theorem 7.3

Suppose that $\dot{\sigma} = ((\gamma, \tau), \rho)$ and that $\mathit{last}\,\phi = \ell$. Then

$\mathit{EvalFrom}\,\ell\,\dot{\sigma} = \mathit{Root}\,\tau\,(\mathit{Eval}\,((\gamma, \ell), \rho))$ where

$\mathit{Root}\,\tau\,((\gamma, \ell), \rho) = ((\gamma, \tau), \rho)$

But we may expand *Eval* to $\mathit{fix}(H)$ where

$H = \lambda\kappa\lambda\sigma.\mathit{Done}\,(\sigma) \longrightarrow \sigma,\ \kappa(\mathit{Step}\,(\sigma))$.

This can now be proved by dual fixpoint induction.

Base Case $\mathit{Root}\,\tau\,(\bot\,\dot{\sigma}) = \bot \cong \bot = \bot\,\dot{\sigma}$

Inductive Step Suppose that $\dot{\sigma}_\ell = \mathit{Root}\,\ell\,\dot{\sigma}$ and that for all $\dot{\sigma}$ witb $\dot{\sigma} \cong \dot{\sigma}$: $\mathit{Root}\,\tau\,(\dot{\kappa}\,\dot{\sigma}_\ell) \cong \dot{\kappa}\,\dot{\sigma}$.

We wish to demonstrate that $\mathit{Root}\,\tau\,(H\,\dot{\kappa}\,\dot{\sigma}_\ell) \cong \underline{\mathit{unwind}}_1\,\dot{\kappa}\,\dot{\sigma}$.

We consider four cases for $\mathit{Done}(\dot{\sigma}_\ell)$:

True Then $H\,\acute{\kappa}\,\acute{\sigma}_\ell = \acute{\sigma}_\ell$. If $Spine\,(fst(\acute{\sigma}_\ell)) = \phi$ then

$$fix(\underline{unwind}'_1)\,\grave{\sigma} = \grave{\sigma}' = (o, \phi, \psi, \gamma, \rho, \delta)$$

by Lemma 7.4. Therefore

$$\underline{unwind}_1\,\grave{\kappa}\,\grave{\sigma} = \underline{entry}\,n\,(\grave{\kappa} \circ \underline{pop}n \circ \underline{step})\,\grave{\sigma}'$$

But $\#\,\phi < n + 1$, so this becomes $\grave{\sigma}'$. Therefore $Root\,r\,\acute{\sigma}_\ell = \acute{\sigma} \cong \grave{\sigma}'$, and because $\grave{\sigma}$ and $\grave{\sigma}'$ differ only to the extent that the stack is unwound, we have the required result.

False We first observe that the left hand side of the equation is $Root\,r\,(\acute{\kappa}(Step'(\acute{\sigma}_\ell)))$. Next we discover that the right hand side is

$$(\grave{\kappa} \circ \underline{pop}n \circ \underline{step})\,\grave{\sigma}'$$

But this is $\grave{\kappa}\,\grave{\sigma}''$ where

$$\begin{aligned}\grave{\sigma}'' &= \underline{pop}n \circ \underline{step}\,\grave{\sigma}' \\ &= \underline{pop}n \circ \underline{step}\,(o, \ell_f : \phi, \psi, \gamma, \rho, \delta) \\ &= \underline{pop}n\,(o, \phi, \psi, \gamma', \rho, \delta) \\ &= (o, \phi', \psi, \gamma', \rho, \delta)\end{aligned}$$

And $\phi' = drop\,(n-1)\,\phi$ and $((\gamma', \ell), \rho) = Step'\,(\acute{\sigma}_\ell)$.
Hence $Root\,r\,(Step'(\acute{\sigma}_\ell)) \cong \underline{step}\,\grave{\sigma}'$ And from this we have result, by induction.

$\stackrel{?}{=}$ In this case $\acute{\sigma} = \stackrel{?}{=}$ and $\grave{\sigma} = \stackrel{?}{=}$, which implies $Root\,r\,(H\,\acute{\kappa}\,\acute{\sigma}) = \stackrel{?}{=}$ and $\underline{unwind}_1\,\grave{\kappa}\,\grave{\sigma} = \stackrel{?}{=}$.

$\perp$ This condition occurs when $\acute{\sigma} = \perp$ or when the inner loop of the operational semantics fails to terminate. In the first case $\grave{\sigma} = \perp$ with the result that $Root\,r\,(H\,\acute{\kappa}\,\acute{\sigma}) = \perp$ and $\underline{unwind}_1\,\grave{\kappa}\,\grave{\sigma} = \perp$. In the second case we may use Lemma 7.4 to show that inner loop failure to terminate occurs in both semantics at the same time.

□

We are now able to demonstrate, as a simple corollary, that the $\underline{eval}$ instruction is equivalent to performing an $EvalFrom\,\ell$ where ℓ is the element on top of the stack.

Corollary 7.5

If $\acute{\sigma} \cong \grave{\sigma}$, *with* $\grave{\sigma} = (o, \ell : \phi, \psi, \gamma, \rho, \delta)$, *then*

$$EvalFrom\,\ell\,\acute{\sigma} \cong \underline{eval}(\grave{\sigma}).$$

$$
\begin{array}{lcl}
\underline{unwind_2}\,\sigma & = & (\nu \in \mathrm{Ide}) \longrightarrow \underline{entry}\,n\,\kappa'\,\sigma',\ \sigma' \\
 & & \quad \text{where } \kappa' = \underline{unwind_2} \circ \underline{pop}\,n \circ \underline{step} \\
 & & \qquad\ n = ArgsGM\,(\nu \mid \mathrm{Ide})\,\rho \\
 & & \qquad\ \nu = \gamma\,\ell \\
 & & \qquad\ \sigma' = \mathit{fix}(\underline{unwind_1'})\,\sigma \\
 & & \qquad\ (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma'
\end{array}
$$

$$
\begin{array}{lcl}
\underline{unwind_3}\,\sigma & = & (\nu \in \mathrm{Ide}) \longrightarrow \underline{entry}\,n\,\kappa'\,\sigma',\ \sigma' \\
 & & \quad \text{where } \kappa' = \underline{unwind_3} \circ \underline{pop}\,n \circ \underline{step} \circ \underline{unpack}\,n \\
 & & \qquad\ n = ArgsGM\,(\nu \mid \mathrm{Ide})\,\rho \\
 & & \qquad\ \nu = \gamma\,\ell \\
 & & \qquad\ \sigma' = \mathit{fix}(\underline{unwind_1'})\,\sigma \\
 & & \qquad\ (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma'
\end{array}
$$

Figure 7.7: The $\underline{unwind_2}$ and $\underline{unwind_3}$ Instructions

The proof is a simple re-arrangement of the equations defining *restore* and *eval*.

Proof of Corollary 7.5

By the definition of $\underline{eval}(\dot{\sigma})$ as
$(\underline{restore} \circ \mathit{fix}(\underline{unwind_1}))\,(o, [\ell], \psi, \gamma, \rho, (\phi : \delta))$,
and the definition of *restore*.

□

Finally we may perform some rearrangement to the *unwind* instruction to eliminate the explicit fixpoint, and to unpack the arguments to the function. Let us define $\underline{unwind_2} = \mathit{fix}(\underline{unwind_1})$. Then we observe (by expanding the fixpoint) that $\underline{unwind_2} = \underline{unwind_1}\,\underline{unwind_2}$, which by substituting into the definition of $\underline{unwind_1}$ allows us to deduce the alternative definition of $\underline{unwind_2}$ given in Figure 7.7.

Informally we are substituting a constant continuation for κ in the definition of $\underline{unwind_1}$. This is important because we will find that κ is indeed constant during execution of the program. This contrasts with the situation during the proof of Theorem 7.3, where we were successively constructing better approximations to $\underline{unwind_2}$.

Finally, to produce an $\underline{unwind_3}$ instruction that mimics the *unwind* instruction described in [Peyton Jones 87, page 323], we must unpack the arguments from the vertibræ. This operation is performed by the *unpack* instruction. It is allowed because *step* does not currently access its arguments from the stack, but instead recomputes the locations of its arguments at each step.

We now consider ways to adapt this definition into something that is nearer to that of the G-machine.

7.3 Continuations From Combinators

In this section we will perform a series of transformations to the stack semantics we have derived in Section 7.2. We first demonstrate how the continuations associated with each combinator in the program text may be represented by the composition of a small number of simple instructions. This fixed set of instructions, which we represent as combinators, will correspond to the instruction set of the abstract machine we will derive.

We wish to eliminate the use of *Step'* from our stack semantics. Our approach will be to show that in every case the continuation corresponding to *Step'* can be constructed by composing combinators from the abstract machines architecture. These instructions will work directly with the G-machine state, rather than invoking *Step'*.

To do this we define a new *unwind* instruction called $\underline{unwind_4}$. This appears in Figure 7.8. This instruction generates the sequence

$$\underline{update}\, n \circ \kappa''$$

in place of *step*. To do this a compilation of the combinator Γ takes place for user defined functions. This compilation is defined in Figure 7.9. The auxiliary functions that are used are referred to as instructions and they are defined in Figure 7.10. We must show that the two continuations are the same. This is done in Theorem 7.6.

Theorem 7.6

For all σ *in* S*:*

$$\underline{unwind_3}\, \sigma = \underline{unwind_4}\, \sigma.$$

$$
\begin{array}{lcll}
\underline{unwind_4}\ \sigma = (\nu \in \mathrm{Ide}) \longrightarrow \underline{entry}\, n\, \kappa'\, \sigma',\ \sigma' & & & \\
\quad \text{where } \kappa' & = & \underline{unwind_4} \circ \underline{pop}\, n \circ \underline{update}\, n \circ \kappa'' \circ \underline{unpack}\, n & \\
\quad\quad n & = & ArgsGM\ \mathrm{I}\ \rho & \\
\quad\quad \nu & = & \gamma\ \ell & \\
\quad\quad \mathrm{I} & = & \nu \upharpoonright \mathrm{Ide} & \\
\quad\quad \sigma' & = & fix(\underline{unwind_1'})\,\sigma & \\
\quad\quad (o, \ell : \phi, \psi, \gamma, \rho, \delta) & = & \sigma' & \\
\quad\quad \kappa'' & = & \mathrm{I} = [\![\mathrm{null}]\!] & \longrightarrow\ \underline{null} \circ \underline{eval} \circ \underline{push}\,0, \\
 & & \mathrm{I} = [\![\mathrm{head}]\!] & \longrightarrow\ \underline{hd} \circ \underline{eval} \circ \underline{push}\,0, \\
 & & \mathrm{I} = [\![\mathrm{cons}]\!] & \longrightarrow\ \underline{cons} \circ \underline{push}\,1 \circ \underline{push}\,1, \\
 & & \mathrm{I} = [\![+]\!] & \longrightarrow\ \underline{add} \circ \underline{eval} \circ \underline{push}\,2 \\
 & & & \qquad \circ \underline{eval} \circ \underline{push}\,0, \\
 & & \mathrm{I} = [\![=]\!] & \longrightarrow\ \underline{eq} \circ \underline{eval} \circ \underline{push}\,2 \\
 & & & \qquad \circ \underline{eval} \circ \underline{push}\,0, \\
 & & \mathrm{I} = [\![\mathrm{if}]\!] & \longrightarrow\ (\underline{if}(\underline{push}\,1)(\underline{push}\,2)) \\
 & & & \qquad \circ \underline{eval} \circ \underline{push}\,0, \\
 & & & \mathcal{C}\,(\rho\,\mathrm{I})\,\beta_{\mathrm{arid}}\,0
\end{array}
$$

Figure 7.8: The $\underline{unwind_4}$ Instruction

We first consider the case of the built-in functions, because for each of them κ'' is a constant. We can state the required equivalence formally as Theorem 7.7. The built-in function instructions are defined in Figure 7.11.

Theorem 7.7

$$
\begin{array}{llllll}
\mathit{If}\ \mathrm{I} = [\![\mathrm{null}]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,1 \circ \underline{null} \circ \underline{eval} \circ \underline{push}\,0 \\
\mathit{If}\ \mathrm{I} = [\![\mathrm{head}]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,1 \circ \underline{hd} \circ \underline{eval} \circ \underline{push}\,0 \\
\mathit{If}\ \mathrm{I} = [\![\mathrm{cons}]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,2 \circ \underline{cons} \circ \underline{push}\,1 \circ \underline{push}\,1 \\
\mathit{If}\ \mathrm{I} = [\![+]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,2 \circ \underline{add} \circ \underline{eval} \circ \underline{push}\,2 \\
 & & & & \qquad \circ \underline{eval} \circ \underline{push}\,0 \\
\mathit{If}\ \mathrm{I} = [\![=]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,2 \circ \underline{eq} \circ \underline{eval} \circ \underline{push}\,2 \\
 & & & & \qquad \circ \underline{eval} \circ \underline{push}\,0 \\
\mathit{If}\ \mathrm{I} = [\![\mathrm{if}]\!] & \mathit{then} & \underline{step} & = & \underline{update}\,3 \circ (\underline{if}(\underline{push}\,1)(\underline{push}\,2)) \\
 & & & & \qquad \circ \underline{eval} \circ \underline{push}\,0
\end{array}
$$

Before we can prove this theorem, we require the following results for argument evaluation. They are both established by rearranging the equations defining *EvalArg*, and applying the result of Corollary 7.5.

$$
\begin{array}{l}
\mathcal{C} : \mathrm{Comb} \to \mathbf{B} \to \mathbf{Z} \to \mathbf{K} \\
\mathcal{C}\,[\![\lambda \mathrm{I}.\Gamma]\!]\,\beta\,n \;=\; \mathcal{C}\,[\![\Gamma]\!]\,(\beta \oplus \{\mathrm{I} \mapsto n\})\,(n+1) \\
\mathcal{C}\,[\![\lambda \mathrm{I}.\mathrm{E}]\!]\,\beta\,n \;=\; \mathcal{E}\,[\![\mathrm{E}]\!]\,(\beta \oplus \{\mathrm{I} \mapsto n\}) \\
\\
\mathcal{E} : \mathrm{Exp} \to \mathbf{B} \to \mathbf{K} \\
\mathcal{E}\,[\![\mathrm{I}]\!]\,\beta \;=\; (\mathrm{I} \in \mathit{dom}(\beta)) \longrightarrow \underline{\mathit{push}}\,(\beta\,[\![\mathrm{I}]\!]),\ \underline{\mathit{pushvalue}}\,([\![\mathrm{I}]\!]\ \mathbf{in}\ \mathrm{N}) \\
\mathcal{E}\,[\![\mathrm{B}]\!]\,\beta \;=\; \underline{\mathit{pushvalue}}\,(\mathcal{B}\,[\![\mathrm{B}]\!]\ \mathbf{in}\ \mathrm{N}) \\
\mathcal{E}\,[\![\mathrm{E}_0\,(\mathrm{E}_1)]\!]\,\beta \;=\; \underline{\mathit{mkap}} \circ (\mathcal{E}\,[\![\mathrm{E}_0]\!]\,(\lambda x.x + 1 \circ \beta)) \circ (\mathcal{E}\,[\![\mathrm{E}_1]\!]\,\beta)
\end{array}
$$

Figure 7.9: Stack Semantic Functions $\mathcal{C}$ and $\mathcal{E}$

$$
\begin{array}{lcl}
\underline{\mathit{pushvalue}}\,\nu\,\sigma & = & (o, \ell : \phi, \psi, \gamma \oplus \{\ell \mapsto \nu\}, \rho, \delta) \\
 & & \quad \text{where } \ell \;=\; \mathit{New}(\gamma) \\
 & & \quad\quad (o, \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{\mathit{push}}\,n\,\sigma & = & (o, (\phi\,!\,n) : \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } (o, \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{\mathit{mkap}}\,\sigma & = & \underline{\mathit{pushvalue}}\,((\ell_0, \ell_1)\ \mathbf{in}\ \mathrm{N})\,(o, \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } (o, \ell_0 : \ell_1 : \phi, \psi, \gamma, \rho, \delta) = \sigma
\end{array}
$$

Figure 7.10: Instructions to Compile User Defined Combinators

Lemma 7.8

If $\dot{o} \cong \dot{o}' = (o, \phi, \psi, \gamma, \rho, \delta)$, *and* $\ell_i = \phi\,!\,(i-1)$ *then*

$$\mathit{EvalArg}\,\ell_i\,\dot{o} \cong (\underline{\mathit{eval}} \circ \underline{\mathit{push}}\,(i-1))\dot{o}'.$$

Proof of Lemma 7.8

This is demonstrated by expanding the $\underline{\mathit{push}}$ instruction and then applying Corollary 7.5.

□

Lemma 7.9

$$
\begin{array}{lcl}
\underline{null}\sigma & = & \underline{pushvalue}\,(\nu = nil)\,(o, \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } \nu = \gamma\,(Elide\,\gamma\,\ell) \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{hd}\,\sigma & = & \nu \in \mathbf{C} \longrightarrow (o, fst\,(\nu \mid \mathbf{C}) : \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \\
 & & \quad \text{where } \nu = \gamma\,(Elide\,\gamma\,\ell) \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{cons}\sigma & = & \underline{pushvalue}\,((\ell_0, \ell_1)\ \mathbf{in}\ \mathbf{N})\,(o, \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } (o, \ell_0 : \ell_1 : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{add}\,\sigma & = & (\nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z}) \longrightarrow \underline{pushvalue}\,\nu\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \\
 & & \quad \text{where } \nu = (\nu_0 \mid \mathbf{Z} + \nu_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{N} \\
 & & \qquad \nu_i = \gamma\,(Elide\,\gamma\,\ell_i) \\
 & & \qquad (o, \ell_1 : \ell_0 : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{eq}\,\sigma & = & (\nu_0 \in \mathbf{Z} \wedge \nu_1 \in \mathbf{Z}) \longrightarrow \underline{pushvalue}\,\nu\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \\
 & & \quad \text{where } \nu = (\nu_0 \mid \mathbf{Z} = \nu_1 \mid \mathbf{Z})\ \mathbf{in}\ \mathbf{N} \\
 & & \qquad \nu_i = \gamma\,(Elide\,\gamma\,\ell_i) \\
 & & \qquad (o, \ell_1 : \ell_0 : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
 & & \\
\underline{if}\kappa_T\kappa_F\,\sigma & = & \nu \in \mathbf{T} \longrightarrow (\nu \mid \mathbf{T} \longrightarrow \kappa_T, \kappa_F)\,(o, \phi, \psi, \gamma, \rho, \delta), \stackrel{?}{=} \\
 & & \quad \text{where } \nu = \gamma\,(Elide\,\gamma\,\ell) \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma
\end{array}
$$

Figure 7.11: Instructions for Built-in Functions

If $\dot{\sigma} \cong \dot{\sigma}' = (o, \phi, \psi, \gamma, \rho, \delta)$, *and* $\ell_i = \phi\,!\,(i-1)$ *then*

$$(EvalArg\,\ell_2 \circ EvalArg\,\ell_1)\dot{\sigma} \cong (\underline{eval} \circ \underline{push}2 \circ \underline{eval} \circ \underline{push}1)\dot{\sigma}'$$

Proof of Lemma 7.9

This is demonstrated by expanding the *push* instructions and then applying Corollary 7.5 twice.

□

We are now able to establish the result of Theorem 7.7.

Proof of Theorem 7.7

By cases of the built-in function.

[[null]] By the definition of *Step'* as $NullStep \circ EvalArg\,\ell_1$, we may use Lemma 7.8 to deduce that

$$\dot{\sigma}' = EvalArg\,\ell_1\,\dot{\sigma} \cong \dot{\sigma}'' = (\underline{eval} \circ \underline{push}\,0)\,\dot{\sigma}'.$$

But $NullStep\,\dot{\sigma}' \cong (\underline{update}\,1 \circ \underline{null})\,\dot{\sigma}''$.

[[head]] This case is proved in a similar way to that of [[null]].

[[cons]] This follows directly from the definition of $\underline{cons}$ and *ConsStep*.

[[+]] By the definition of *Step'* as $PlusStep \circ EvalArg\,\ell_2 \circ EvalArg\,\ell_1$, we may use Lemma 7.9 to deduce that

$$\dot{\sigma}' = EvalArg\,\ell_2 \circ EvalArg\,\ell_1\,\dot{\sigma} \cong \dot{\sigma}'',$$

where $\dot{\sigma}'' = (\underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0)\,\dot{\sigma}'$. But

$$PlusStep\,\dot{\sigma}' \cong (\underline{update}\,2 \circ \underline{add})\,\dot{\sigma}''.$$

[[=]] This case is proved in a similar way to that of [[+]].

[[if]] This case is proved in a similar way to that of [[null]].

□

Finally we wish to show that an analogous transformation to that of the previous step may be performed for user defined functions. Before we do this we define a predicate b which allows us to compare the local bindings for variables from the two implementations. The original operational semantics bound local variables to the application nodes where the relevant argument was. The G-machine binding $\hat{\beta}$ binds an identifier to a stack offset. For a given γ and ϕ we may therefore express the equivalence of the bindings using $b_{(\gamma,\phi)}(\hat{\beta})$, which is defined in Definition 7.10.

Definition 7.10

For any γ and ϕ define

$$b_{(\gamma,\phi)}(\hat{\beta}) \Leftrightarrow \forall \mathrm{I}.\, Arg\,(\gamma, \hat{\beta}\,[\![\mathrm{I}]\!]) = \phi\,!\,(\hat{\beta}\,[\![\mathrm{I}]\!])$$

We now relate the combinator compilations, $\acute{\mathcal{C}}$ and $\grave{\mathcal{C}}$, in the two implementations, using Theorem 7.11.

Theorem 7.11

Suppose that $(o, \ell : \phi, \psi, \gamma, \rho, \delta) = fix(\underline{unwind}'_1)\,\sigma$. *Assume that* $\gamma\,\ell \in$ Ide *with* $\mathrm{I} = \gamma\,\ell \mid \mathrm{Ide}$ *in* $dom(\rho)$ *and let* $\Gamma = \rho(\mathrm{I})$. *Let* $n = ArgsGM\,\mathrm{I}\,\rho$ *and* $\sigma' = (o, \phi', \psi, \gamma, \rho, \delta) = \underline{unpack}n\,(o, \phi, \psi, \gamma, \rho, \delta)$.

Then, for all Γ *in* Comb, *all* $\hat{\beta}$ *in* $\hat{\mathbf{B}}$ *with* $b_{(\gamma,\phi)}(\hat{\beta})$ *and all* m *with* $0 \leq m < n$:

$$\acute{\mathcal{C}}\,[\![\Gamma]\!]\,\hat{\beta}\,(drop\,m\,\phi)\,(\gamma, \ell) = (\gamma', \ell')$$

if, and only if

$$\grave{\mathcal{C}}\,[\![\Gamma]\!]\,\hat{\beta}\,m\,\grave{\sigma}' = (o, \ell' : \phi', \psi, \gamma', \rho, \delta).$$

The proof obviously involves a structural induction. One of the components of the syntax of Comb is an expression in E, and we will need a subsidiary result for this syntactic object as well. This is given as Theorem 7.12. The theorem states that for congruent states, with equivalent bindings $\hat{\beta}$, the expression compilations $\acute{\mathcal{E}}$ and $\grave{\mathcal{E}}$ transform the initial states to new congruent states.

Theorem 7.12

For any $\grave{\sigma}' = (o, \phi, \psi, \gamma, \rho, \delta)$, E *in* Exp, *and* $\hat{\beta}$ *in* $\hat{\mathbf{B}}$ *with* $b_{(\gamma,\phi)}(\hat{\beta})$:

$$\acute{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\hat{\beta}\,(\gamma, \ell) = (\gamma', \ell')$$

if, and only if

$$\grave{\mathcal{E}}\,[\![\mathrm{E}]\!]\,\hat{\beta}\,\grave{\sigma}' = (o, \ell' : \phi, \psi, \gamma', \rho, \delta).$$

This result also requires a structural induction to complete the proof.

Proof of Theorem 7.12

By a structural induction on E.

$[\![\mathrm{I}]\!]$ There are two cases to consider: either I is a local variable or it is a global function. We consider them separately.

1. If I is a local variable it occurs in $dom(\grave{\beta})$ and hence also in $dom(\acute{\beta})$. We observe that $\acute{\mathcal{E}}[\![\mathrm{I}]\!]\acute{\beta}(\gamma, \ell)$ is $(\gamma, Arg(\gamma, \acute{\beta}[\![\mathrm{I}]\!]))$. If we now consider $\grave{\mathcal{E}}$, we observe that:
$$\grave{\mathcal{E}}[\![\mathrm{I}]\!]\grave{\beta}\grave{\sigma}' = \underline{push}(\grave{\beta}[\![\mathrm{I}]\!])\grave{\sigma}',$$
which is $(o, \phi ! \grave{\beta}[\![\mathrm{I}]\!] : \phi, \psi, \gamma, \rho, \delta)$. But
$$Arg(\gamma, \acute{\beta}[\![\mathrm{I}]\!]) = \phi ! \grave{\beta}[\![\mathrm{I}]\!].$$
2. If I is not in $dom(\grave{\beta})$, and hence also not in $dom(\acute{\beta})$, then $\acute{\mathcal{E}}[\![\mathrm{I}]\!]\acute{\beta}(\gamma, \ell)$ is $(\gamma \oplus \{\ell' \mapsto [\![\mathrm{I}]\!]\ \mathbf{in}\ \mathrm{N}\}, \ell')$, where $\ell' = New(\gamma)$. Also $\grave{\mathcal{E}}[\![\mathrm{I}]\!]\grave{\beta}\grave{\sigma}' = \underline{pushvalue}([\![\mathrm{I}]\!]\ \mathbf{in}\ \mathrm{N})\grave{\sigma}'$, which is
$$(o, \ell' : \phi, \psi, \gamma \oplus \{\ell' \mapsto [\![\mathrm{I}]\!]\ \mathbf{in}\ \mathrm{N}\}, \rho, \delta).$$

$[\![\mathrm{B}]\!]$ We observe that $\acute{\mathcal{E}}[\![\mathrm{B}]\!]\acute{\beta}(\gamma, \ell)$ is $(\gamma \oplus \{\ell' \mapsto (\mathcal{B}[\![\mathrm{B}]\!])\ \mathbf{in}\ \mathrm{N}\}, \ell')$, where $\ell' = New(\gamma)$. Also $\grave{\mathcal{E}}[\![\mathrm{B}]\!]\grave{\beta}\grave{\sigma}' = \underline{pushvalue}(\mathcal{B}[\![\mathrm{B}]\!]\ \mathbf{in}\ \mathrm{N})\grave{\sigma}'$ which is
$$(o, \ell' : \phi, \psi, \gamma \oplus \{\ell' \mapsto (\mathcal{B}[\![\mathrm{B}]\!])\ \mathbf{in}\ \mathrm{N}\}, \rho, \delta).$$

$[\![\mathrm{E_0}(\mathrm{E_1})]\!]$ We first observe that $\acute{\mathcal{E}}[\![\mathrm{E_0}(\mathrm{E_1})]\!]\acute{\beta}(\gamma_2, \ell_2)$ is
$$(\gamma_0 \oplus \{\ell' \mapsto (\ell_1, \ell_2)\ \mathbf{in}\ \mathrm{N}\}, \ell'),$$
where $(\gamma_i, \ell_i) = \acute{\mathcal{E}}[\![\mathrm{E}_i]\!]\acute{\beta}(\gamma_{i+1}, \ell_{i+1})$, for $i = 0, 1$ and $\ell' = New(\gamma_0)$.
Also $\grave{\mathcal{E}}[\![\mathrm{E_0}(\mathrm{E_1})]\!]\grave{\beta} = \underline{mkap} \circ (\grave{\mathcal{E}}[\![\mathrm{E_0}]\!](\lambda x.x + 1 \circ \grave{\beta})) \circ (\grave{\mathcal{E}}[\![\mathrm{E_1}]\!]\grave{\beta})$. Assume inductively that the theorem holds for $[\![\mathrm{E_0}]\!]$ and $[\![\mathrm{E_1}]\!]$. In particular, therefore,
$$\acute{\mathcal{E}}[\![\mathrm{E_1}]\!]\acute{\beta}(\gamma_2, \ell_2) = (\gamma_1, \ell_1)$$
and
$$\grave{\mathcal{E}}[\![\mathrm{E_1}]\!]\grave{\beta}(o, \phi, \psi, \gamma_2, \rho, \delta) = (o, \ell_1 : \phi, \psi, \gamma_1, \rho, \delta).$$
We now notice that $b_{(\gamma,\phi)}(\hat{\beta}) \Leftrightarrow b_{(\gamma,(\ell:\phi))}(\hat{\beta}, (\lambda x.x + 1 \circ \hat{\beta}))$, and hence we may use the inductive hypothesis to conclude that
$$\acute{\mathcal{E}}[\![\mathrm{E_0}]\!]\acute{\beta}(\gamma_1, \ell_1) = (\gamma_0, \ell_0).$$
and
$$\grave{\mathcal{E}}[\![\mathrm{E_0}]\!]\grave{\beta}(o, \ell_1 : \phi, \psi, \gamma_1, \rho, \delta) = (o, \ell_0 : \ell_1 : \phi, \psi, \gamma_0, \rho, \delta).$$

But
$\acute{\mathcal{E}}\,[\![\mathrm{E_0}\,(\mathrm{E_1})]\!]\,\acute{\beta}\,(\gamma_2, \ell_2) = (\gamma_0 \oplus \{\ell \mapsto (\ell_0, \ell_1)\ \mathbf{in}\ \mathrm{N}\}, \ell)$ where
$\ell = \mathit{New}(\gamma_0)$. And
$\grave{\mathcal{E}}\,[\![\mathrm{E_0}\,(\mathrm{E_1})]\!]\,\grave{\beta}\,(o, \phi, \psi, \gamma_2, \rho, \delta) = \underline{\mathit{mkap}}(o, \ell_0 : \ell_1 : \phi, \psi, \gamma_0, \rho, \delta)$
which is

$$(o, \ell : \phi, \psi, \gamma_0 \oplus \{\ell \mapsto (\ell_0, \ell_1)\ \mathbf{in}\ \mathrm{N}\}, \rho, \delta),$$

where $\ell = \mathit{New}(\gamma_0)$.

□

We may now continue to prove that the combinator compilation functions $\acute{\mathcal{C}}$ and $\grave{\mathcal{C}}$ are equivalent.

Proof of Theorem 7.11

A structural induction on Γ.

[λI.Γ] Observe that

$$\acute{\mathcal{C}}\,[\![\lambda\mathrm{I}.\Gamma]\!]\,\acute{\beta}\,(\mathit{drop}\,m\,\phi) = \acute{\mathcal{C}}\,[\![\Gamma]\!]\,\acute{\beta}'\,(\mathit{drop}\,(m+1)\,\phi)$$

where $\acute{\beta}' = \acute{\beta} \oplus \{\mathrm{I} \mapsto \phi\,!\,m\}$. However

$$\grave{\mathcal{C}}\,[\![\lambda\mathrm{I}.\Gamma]\!]\,\grave{\beta}\,m\,\sigma' = \grave{\mathcal{C}}\,[\![\Gamma]\!]\,\grave{\beta}'\,(m+1)\,\sigma'$$

where $\grave{\beta}' = \grave{\beta} \oplus \{\mathrm{I} \mapsto m\}$. But $b_{(\gamma,\phi)}\,(\hat{\beta}')$, and so by inductive hypothesis we conclude the required result.

[λI.E] Observe that

$$\acute{\mathcal{C}}\,[\![\lambda\mathrm{I}.\mathrm{E}]\!]\,\acute{\beta}\,(\mathit{drop}\,m\,\phi) = \acute{\mathcal{C}}\,[\![\mathrm{E}]\!]\,\acute{\beta}'\,(\mathit{drop}\,(m+1)\,\phi)$$

where $\acute{\beta}' = \acute{\beta} \oplus \{\mathrm{I} \mapsto \phi\,!\,m\}$. However

$$\grave{\mathcal{C}}\,[\![\lambda\mathrm{I}.\mathrm{E}]\!]\,\grave{\beta}\,m\,\sigma' = \grave{\mathcal{C}}\,[\![\mathrm{E}]\!]\,\grave{\beta}'\,(m+1)\,\sigma'$$

where $\grave{\beta}' = \grave{\beta} \oplus \{\mathrm{I} \mapsto m\}$. But $b_{(\gamma,\phi)}\,(\hat{\beta}')$, so we conclude that the base case of our induction holds, by appealing to Theorem 7.12.

□

This completes the proof that the user defined functions are compiled in the same way. Together with the result of Theorem 7.7, we are able to demonstrate that Theorem 7.6 holds.

$$
\begin{array}{lcl}
\underline{unwind}\sigma & = & (\nu \in \mathrm{Ide}) \longrightarrow \rho(\nu \mid \mathrm{Ide})\,\sigma',\ \sigma' \\
 & & \quad \text{where } \nu = \gamma\,\ell \\
 & & \qquad \sigma' = \mathit{fix}(\underline{unwind}'_1)\,\sigma \\
 & & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma' \\
 & & \\
\underline{exit}\,n & = & \underline{unwind} \circ \underline{pop}\,n \circ \underline{update}\,n
\end{array}
$$

Figure 7.12: The $\underline{unwind}$ Instruction

$$
\begin{array}{lcl}
\mathcal{D} : \mathrm{Defs} \to \mathbf{U} & & \\
\mathcal{D}\,[\![\Delta_0 \text{ and } \Delta_1]\!] & = & \mathcal{D}\,[\![\Delta_0]\!] \oplus \mathcal{D}\,[\![\Delta_1]\!] \\
\mathcal{D}\,[\![\mathrm{I} = \Gamma]\!] & = & \{\mathrm{I} \mapsto \underline{entry}\,n\,\kappa\} \\
 & & \quad \text{where } n = \mathit{Args}\,[\![\Gamma]\!] \\
 & & \qquad \kappa = \underline{exit}\,n \circ \mathcal{C}\,[\![\Gamma]\!]\,\{\}\,0
\end{array}
$$

Figure 7.13: The Stack Semantic Function $\mathcal{D}$

$$
\begin{array}{lcl}
\rho_{\mathrm{init}}\,[\![\mathrm{null}]\!] & = & \underline{entry}\,1(\underline{exit}\,1 \circ \underline{null} \circ \underline{eval} \circ \underline{push}\,0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{hd}]\!] & = & \underline{entry}\,1(\underline{exit}\,1 \circ \underline{hd} \circ \underline{eval} \circ \underline{push}\,0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{cons}]\!] & = & \underline{entry}\,2(\underline{exit}\,2 \circ \underline{cons} \circ \underline{push}\,1 \circ \underline{push}\,1) \\
\rho_{\mathrm{init}}\,[\![\mathrm{add}]\!] & = & \underline{entry}\,2(\underline{exit}\,2 \circ \underline{add} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{eq}]\!] & = & \underline{entry}\,2(\underline{exit}\,2 \circ \underline{eq} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0) \\
\rho_{\mathrm{init}}\,[\![\mathrm{if}]\!] & = & \underline{entry}\,3(\underline{exit}\,3 \circ \underline{if}(\underline{push}\,1)(\underline{push}\,2) \circ \underline{eval} \circ \underline{push}\,0)
\end{array}
$$

Figure 7.14: The Initial Environment ρ_{init}

Proof of Theorem 7.6

The proof is effectively a proof by fixpoint induction, although as there are no longer explicit fixpoints in the definitions of either $\underline{unwind}_3$ or $\underline{unwind}_4$, we will use an equational reasoning argument to establish the required result.

Assume that $\sigma' = \mathit{fix}(\underline{unwind}'_1)\,\sigma = (o, \ell : \phi, \psi, \gamma, \rho, \delta)$, and let $\nu = \gamma\ell$. If $\nu \notin \mathrm{Ide}$ then trivially $\underline{unwind}_3 = \underline{unwind}_4$. Alternatively assume that $\nu \in \mathrm{Ide}$, and that $\mathit{ArgsGM}\,(\nu \mid \mathrm{Ide})\,\rho = n$. If $\#\,\phi < n$ then again the result is trivial.

Assume therefore that $\#\,\phi \geq n$. Then $\sigma'' = \underline{unpack}\,n\,\sigma'$. We must now show that

$\underline{step}\,\sigma'' = \underline{update}\,n \circ \kappa''$.

1. If the identifier represents a built-in function, then from Theorem 7.7, we know that

   $$\underline{step}\,\sigma'' = \underline{update}\,n \circ \kappa''.$$

2. Otherwise the identifier must be a user defined function, *i.e.* $\kappa'' = \dot{C}\,(\rho\,(\nu \mid \mathrm{Ide}))\,\beta_{\mathrm{arid}}\,0$.
   But $\underline{step}\sigma'' = (o, \phi'', \psi, \gamma', \rho, \delta)$, where $r = last\,\phi''$, and
   $((\gamma', r), \rho) = Step'((\gamma, r), \rho)$.
   By Lemma 7.4, we know that $\ell : \phi = Spine\,(\gamma, r)$, so expanding $Step'$ further, we have

   $$((\gamma', r), \rho) = ((\gamma'' \oplus \{(\phi\,!\,(n-1)) \mapsto \ell'\}, r), \rho),$$

   where

   $$(\gamma'', \ell') = \acute{C}\,(\rho\,(\nu \mid \mathrm{Ide}))\,\beta_{\mathrm{arid}}\,\phi\,(\gamma, \ell).$$

   But this is merely a special case of Theorem 7.11.

□

The major advance that the $\underline{unwind}_4$ instruction has over $\underline{unwind}_3$ is that we may change the domain of the environment, U, from

$$[\mathrm{Ide} \to \mathrm{Comb}]$$

to $[\mathrm{Ide} \to \mathrm{K}]$. When we do this we will need to change the definition of unwinding again, so that it looks up a continuation in the environment. We must also ensure that the relevant continuations are entered into the environment. Figure 7.12 defines the final version of the unwinding instruction. In Figure 7.13 a compilation of the programmers environment Δ is defined. The initial environment is ρ_{init} which provides continuations for the built in functions.

This concludes the derivation of a simple G-machine from our original operational semantics. We make one final modification to the language in the next section, where a printer mechanism is added, before we consider some of the optimizations which make the G-machine efficient.

$$
\begin{array}{lcll}
print(\varepsilon) & = & (\varepsilon = nil \vee \varepsilon \in \mathbf{T}) & \longrightarrow [], \\
 & & (\varepsilon \in \mathbf{Z}) & \longrightarrow [\varepsilon \mid \mathbf{Z}], \\
 & & (\varepsilon \in \mathbf{C}) & \longrightarrow prints(\varepsilon \mid \mathbf{C}), \underline{?}
\end{array}
$$

$$
\begin{array}{lcl}
prints(\chi) & = & print(\varepsilon_0) +\!\!+ print(\varepsilon_1) \\
 & & \quad \text{where } (\varepsilon_0, \varepsilon_1) = \chi \mid (\mathbf{E} \times \mathbf{E})
\end{array}
$$

Figure 7.15: A Printer for the Standard Semantics

$$
\begin{array}{lcll}
\underline{print}(o, \ell : \phi, \psi, \gamma, \rho, \delta) = & & & \\
\quad (\nu \in \mathbf{Z}) & & \longrightarrow & (o +\!\!+ (\nu \mid \mathbf{Z}), \phi, \psi, \gamma, \rho, \delta), \\
\quad (\nu \in \mathbf{T} \vee \nu = nil) & & \longrightarrow & (o, \phi, \psi, \gamma, \rho, \delta), \\
\quad (\nu \in \mathbf{C}) & & \longrightarrow & \kappa(o, \ell' : \ell'' : \phi, \psi, \gamma, \rho, \delta)), \underline{?} \\
\quad \text{where } \nu & = & \gamma\, \ell & \\
\quad\quad \kappa & = & \underline{print} \circ \underline{eval} \circ \underline{print} \circ \underline{eval} & \\
\quad\quad (\ell', \ell'') & = & (\nu \mid \mathbf{C}) &
\end{array}
$$

Figure 7.16: The G-machine *print* instruction

7.4 The G-machine Printer Mechanism

To complete the description of the G-machine we must consider how the addition of a printer mechanism affects the semantic definitions of the language. If the result of the program is a basic value we are able to show that the denotational semantics and the G-machine produce the same result. But what happens if we produce a structured data object, such as a list? The denotational semantics is defined to produce the list, but the G-machine will only reduce the initial graph to weak head normal form. Thus we must recursively reduce the elements of the list to weak head normal form. We must also consider these results in the light of the full abstraction problem that was discussed in Chapter 3.

In performing the recursive printing operation, we must consider the possibility that an element of the list is $\perp$. With the G-machine the recursive reduction to WHNF stops, and thus no more reduction occurs. That is, when printing the list $\ell : \perp : \phi$, the G-machine will print ℓ and then halt. This occurs even if the remainder of the list elements in ϕ are all non-$\perp$.

We will therefore have to modify the denotational semantics to ensure that the example above is given the value $\ell : \perp$. We do this by defining *print*

in Figure 7.15.

The corresponding G-machine instruction is defined in Figure 7.16. This is a tail-recursive implementation of the *print* function operating on the G-machine state – a result that we demonstrate in Theorem 7.14 To make the proof easier we shall use the G-machine definition in which the environment is a function from identifiers to the syntactic representation of combinators. This allows us to easily form the derepresentation function. In the modified G-machine with continuations in the environment, we would need an inverse function from K to Comb. This certainly exists, but would be difficult to define.

Definition 7.13

For all $\acute{\epsilon}$ in $\acute{\mathbf{E}}$, with $\acute{\epsilon} \in \mathbf{Z}$, $\acute{\epsilon} \in \mathbf{T}$ or $\acute{\epsilon} \in \mathbf{C}$; we say that

$$\acute{\epsilon} \cong \grave{\sigma}$$

if, and only if,

$$\acute{\epsilon} \sqsupseteq E((\gamma, \mathit{last}\,(\mathit{last}\,(\phi : \delta))), \rho)$$

and

$$e(\acute{\epsilon}, \mathit{Eval}((\gamma, \mathit{last}\,(\mathit{last}\,(\phi : \delta))), \rho)).$$

We now state that the printer mechanisms are congruent.

Theorem 7.14

For all $\acute{\epsilon}$ *in* $\acute{\mathbf{T}}$, $\acute{\mathbf{Z}}$ *and* $\acute{\mathbf{C}}$; *with* $\grave{\sigma}$ *in* $\grave{\mathbf{S}}$ *and* $\acute{\epsilon} \cong \grave{\sigma}$:

$$\mathit{print}\,(\acute{\epsilon}) = \mathit{fst}\,((\underline{\mathit{print}} \circ \underline{\mathit{eval}})\,\grave{\sigma}).$$

The proof is a structural induction on the structure of the output object.

Proof of Theorem 7.14

We first observe that $\acute{\epsilon} \cong \grave{\sigma}$ implies $\acute{\epsilon} \cong \underline{\mathit{eval}}\,\grave{\sigma}$, by the congruence result of Part I. As a consequence of this we need only consider proper values for either semantics result.

($\acute{\varepsilon} = \bot \vee \acute{\varepsilon} = nil \vee \acute{\varepsilon} \in \mathbf{T} \vee \acute{\varepsilon} \in \mathbf{Z}$) In these cases the values of $\gamma\, r$ are respectively $\bot$; *nil* **in** **N**; $\acute{\varepsilon} \mid \mathbf{T}$ and $\acute{\varepsilon} \mid \mathbf{Z}$. And hence the printers produce the same value, *i.e.* $\bot$; []; [] and $[\acute{\varepsilon} \mid \mathbf{Z}]$.

($\varepsilon \in \mathbf{C}$) Suppose that $(r_0, r_1) = \gamma\, r \mid \mathbf{C}$, and
$(\acute{\varepsilon}_0, \acute{\varepsilon}_1) = \chi \mid (\mathbf{E} \times \mathbf{E})$.
Assume inductively that the theorem holds for the respective subcomponents, *i.e.*

$$print\ (\acute{\varepsilon}_i) = fst\ ((\underline{print} \circ \underline{eval})\ \grave{\sigma}_{r_i}),$$

where $\grave{\sigma}_{r_i} = (o, r_i : \phi, \psi, \gamma, \rho, \delta)$. Then

$$print\ (\acute{\varepsilon}) = print\ (\acute{\varepsilon}_0) +\!\!+ print\ (\acute{\varepsilon}_1).$$

But

$$fst\ ((\underline{print} \circ \underline{eval})\ \grave{\sigma}) = fst\ ((\underline{print} \circ \underline{eval})\ \grave{\sigma}_{r_0}) +\!\!+ fst\ ((\underline{print} \circ \underline{eval})\ \grave{\sigma}_{r_1}).$$

□

We are thus able to show that the printer mechanism, which acts as a demand driver for a functional program, is implemented in a congruent way by both the G-machine and the denotational semantics. This concludes our survey of the basic operations of a G-machine.

7.5 Related Work

The G-machine is described in a series of papers [Johnsson 83, Johnsson 84] and Augustsson and Johnssons' theses [Augustsson 87, Johnsson 87] provide the latest, definitive work. Similar material is also presented by Peyton Jones in [1987]. A proof of the material in Sections 7.2 and 7.3 occurs in [Lester 85].

Representing the abstract machine instruction set by combinators is suggested by Wand in [1982]. He also shows how these combinators may be derived from the continuation semantics. A practical use of this technique occurs in [Clinger 84].

7.6 Conclusion

In this chapter we have derived a simple G-machine from the operational semantics. This has been achieved by representing the operational semantics

by sequences of combinators, each of which represents an abstract machine instruction. We recall that since we have established the congruence of Chapter 4, we are able to conclude that this machine correctly implements the language, as specified by the denotational semantics.

In the final chapter we show that a selection of the code improvements, proposed by Peyton Jones [1987], Johnsson [1983] and Augustsson [1987] are correct.

Chapter 8

Store-level Optimizations

In this chapter we establish a stronger form of state equivalence than that of the denotational equivalence given by the derepresentation function E. Under E we are able to equivalence the functions representing quicksort and insertion sort. This is because both functions return a sorted list, and hence are functionally or denotationally equivalent. We would like to be able to talk about the operational equivalence of the evaluation of functions. The intention is that two graphs should satisfy the equivalence, if they are essentially the same. To do this we formally introduce the notion of graph isomorphism.

8.1 Graph, State and Continuation Isomorphism

Informally, two graphs are isomorphic if they have the same structure. The graphs with which we deal were defined in Chapter 2, and we recall that they were finite, labeled, rooted digraphs. Harary gives a general definition of graph isomorphism in [Harary 69, Page 10], which is repeated here. Informally two graphs are isomorphic if they possess the same structure. This is just the property we require when we wish to establish the operational equivalence of two graphs.

Definition 8.1

> Two graphs G and H are isomorphic (written $G \cong H$ or sometimes $G = H$) if there exists a one to one correspondence between their point sets which preserves adjacency.

> [...] It goes without saying that isomorphism is an equivalence relation on graphs.

Two rooted graphs G and H are isomorphic, if in addition, the one to one correspondence maps the root of G to the root of H.

To serve as a model of operational equivalence we would like to include further graphs in the equivalence classes induced by the isomorphism. For example we would like to ignore any indirection nodes that may exist in the graph, and we are not interested when the graphs γ_0 and γ_1 differ only at unreachable nodes. For this reason we will ignore both these features. We first define a function to remove indirection nodes from a graph. This is achieved by generating a map from node labels to node labels. Notice that essential indirection nodes are not elided.

Definition 8.2

$$elide = elide'\,[]$$

where

$$\begin{aligned} elide'\,\ell s\,\gamma\,\ell = {} & (\gamma\,\ell \in \mathbf{I}) \longrightarrow \\ & (((\gamma\,\ell \mid \mathbf{I}) \in \ell s) \longrightarrow \\ & \quad last\,\ell s, \\ & \quad elide'\,((\gamma\,\ell \mid \mathbf{I}) : \ell s)\,\gamma\,(\gamma\,\ell \mid \mathbf{I})), \\ & \ell. \end{aligned}$$

From a given set of root nodes, only part of the graph may be accessible. We define that set $mark\,(\gamma, R)$ to be the set of accessible nodes in γ from R.

Definition 8.3

For a given set of root nodes R in $\mathbf{L}$ and graph γ in $\mathbf{G}$, the set $mark(\gamma, R)$, is defined as follows:

1. If ℓ is a label in R, then ℓ is in the set $mark(\gamma, R)$.
2. If ℓ is in $mark(\gamma, R)$, then let $\nu = \gamma\,\ell$.
   (a) If $\nu \in \mathbf{A}$, then ℓ' and ℓ'' are in $mark(\gamma, R)$, where $(\ell', \ell'') = (\nu \mid \mathbf{A})$.
   (b) If $\nu \in \mathbf{I}$, then ℓ is in $mark(\gamma, R)$, where $\ell = (\nu \mid \mathbf{I})$.

(c) If $\nu \in \mathbf{C}$, then ℓ' and ℓ'' are in $mark(\gamma, R)$, where $(\ell', \ell'') = (\nu \mid \mathbf{C})$.

3. No other labels are in the set $mark(\gamma, R)$.

Using the definition of $mark\,(\gamma, R)$, we may assign $\perp$ to all non-reachable nodes of the graph. This is performed by the function *garbage*.

Definition 8.4

$$
\begin{array}{l}
garbage\ R\,\gamma = \\
\quad \lambda\ell.(\ell \in mark\,(\gamma, R) \longrightarrow \\
\qquad (\ (\gamma\,\ell \in \mathbf{A}) \longrightarrow (\ elide\,\gamma\,(fst\,(\gamma\,\ell \mid \mathbf{A})), \\
\qquad\qquad\qquad\qquad\quad elide\,\gamma\,(snd\,(\gamma\,\ell \mid \mathbf{A}))), \\
\qquad\quad (\gamma\,\ell \in \mathbf{C}) \longrightarrow (\ elide\,\gamma\,(fst\,(\gamma\,\ell \mid \mathbf{C})), \\
\qquad\qquad\qquad\qquad\quad elide\,\gamma\,(snd\,(\gamma\,\ell \mid \mathbf{C}))), \\
\qquad\qquad\qquad\qquad\quad elide\,\gamma\,\ell), \perp).
\end{array}
$$

We are now able to define our extended form of graph-isomorphism, using the function *garbage* and Harary's definition.

Definition 8.5

Let $\gamma_i = garbage\,R_i\,\gamma_i$. Then we say that two graphs, γ_0 and γ_1, are isomorphic, modulo the root sets, R_0 and R_1, if and only if γ_0' is isomorphic to γ_1' under Harary's definition of isomorphism [1969]. We denote this relation by

$$\gamma_0 \textbf{ mod } R_0 \cong \gamma_1 \textbf{ mod } R_1.$$

We are now able to extend this definition of graph-isomorphism, so that it provides a way to compare states and continuations. Informally, two states are isomorphic if we may relabel the nodes in each garbage collected graph to obtain the other.

Definition 8.6

Let $\sigma_i = (o, \phi_i, \psi, \gamma_i, \rho, \delta_i)$ and let R_i be the labels contained in $\phi_i : \delta_i$. The states σ_0 and σ_1 are then state isomorphic, if and only if γ_0 mod $R_0 \cong \gamma_1$ mod R_1 and, if $f : \mathrm{L} \to \mathrm{L}$ relabels nodes in γ_0 with the equivalent node labels in γ_1, then

$$map\,(map\,f)\,(\phi_0 : \delta_0) = (\phi_1 : \delta_1).$$

We write this as

$$\sigma_0 \cong \sigma_1.$$

Similarly, we can extend state isomorphism so that we can compare two continuations. In this case they are continuation-isomorphic if they produce state-isomorphic results when applied to any state.

Definition 8.7

Two continuations, κ and κ' in K, are continuation isomorphic, if and only, if for all σ in S

$$\kappa\,\sigma \cong \kappa'\,\sigma.$$

We are unable to use equality for operational equivalence as this is too strong. In Chapter 6 we were able to use equality to establish the operational equivalence, because we always constructed the same graph in the end. When we move on to consider ways to avoid constructing graphs, we will find that the function *New* may not return the same label for nodes we wish to equivalence.

8.2 Reducing the Amount of Graph Constructed

One of the main observations that Johnsson made in [1983], is that we can transform the code sequences so that less graph is constructed. Consider the conditional expression $[\![\text{if}\,E_0\,E_1\,E_2]\!]$, and suppose we wish to evaluate this. Depending on the result of the evaluation of $[\![E_0]\!]$ we need only construct graph to represent one of $[\![E_1]\!]$ or $[\![E_2]\!]$. Furthermore, this graph will also be immediately evaluated, leading to further possible savings.

To assist in the exposition, we will borrow Johnsson's $\mathcal{E}$-scheme and $\mathcal{R}$-scheme notation. These are defined, in terms of $\mathcal{E}$ from Chapter 7, which correspond to Johusson's $\mathcal{C}$-scheme. We will therefore refer to it as $\mathcal{E}_C$ in this chapter.

Definition 8.8

$$
\begin{array}{lcl}
\underline{call}\,\mathrm{I}\,n\,\sigma & = & (\underline{eval} \circ \underline{slide}\,n \circ \kappa')\,\sigma \\
& & \quad \text{where } \rho[\![\mathrm{I}]\!] = \underline{entry}\,n\,(exit\,n \circ \kappa' \circ \underline{unpack}\,n) \\
& & \qquad (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma \\
\\
\underline{slide}\,n\,\sigma & = & (o, \ell : drop\,n\,\phi, \psi, \gamma, \rho, \delta) \\
& & \quad \text{where } (o, \ell : \phi, \psi, \gamma, \rho, \delta) = \sigma
\end{array}
$$

Figure 8.1: The *call* and *slide* Instructions

$$
\begin{array}{lcl}
\mathcal{E}_R[\![\mathrm{E}]\!]\,\beta\,n & = & \underline{exit}\,n \circ \mathcal{E}[\![\mathrm{E}]\!]\,\beta \\
\mathcal{E}_E[\![\mathrm{E}]\!]\,\beta & = & \underline{eval} \circ \mathcal{E}[\![\mathrm{E}]\!]\,\beta \\
\mathcal{E}_C[\![\mathrm{E}]\!]\,\beta & = & \mathcal{E}[\![\mathrm{E}]\!]\,\beta
\end{array}
$$

To abbreviate some of the notation we define β^{+n}. This allows us to adjust stack offsets in a simple manner.

Definition 8.9

Define

$$\beta^{+n} = (\lambda x.x + n) \circ \beta.$$

The first theorem that we will prove is that we need not construct the spine when we reduce a function I, of arity n, when it is applied to n arguments. To specify the problem exactly, we introduce two new instructions: *call* and *slide* in Figure 8.1. The $\underline{call}\,\mathrm{I}\,n$ instruction applies the function I to n arguments on top of the stack, leaving the result on the top of the stack. The $\underline{slide}\,n$ instruction squeezes out n arguments from the stack and leaves the top of the stack unchanged.

Theorem 8.10

For a function I, *of arity* n,

$$\mathcal{E}_E[\![\mathrm{I}\,\mathrm{E}_1\,\ldots\,\mathrm{E}_n]\!]\,\beta \cong \underline{call}\,\mathrm{I}\,n \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\,\beta^{+(n-1)} \circ \ldots \circ \mathcal{E}_C[\![\mathrm{E}_n]\!]\,\beta.$$

This is proved by substituting for the instructions in the alternative code sequences.

Proof of Theorem 8.10

We first observe $\mathcal{E}_E[\![\mathrm{I}\,\mathrm{E}_1 \ldots \mathrm{E}_n]\!]\beta$ is defined as

$$\underline{eval} \circ \underline{mkap} \circ \ldots \circ \underline{mkap} \circ \underline{pushvalue}(\mathrm{I}\ \mathbf{in}\ \mathbf{N}) \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta^{+(n-1)} \circ \ldots \circ \mathcal{E}_C[\![\mathrm{E}_n]\!]\beta.$$

Therefore, let

$$\begin{aligned} \sigma_1 &= (o, \ell_1 : \ldots : \ell_n : \phi, \psi, \gamma, \rho, \delta) \\ &= (\mathcal{E}_C[\![\mathrm{E}_1]\!]\beta^{+(n-1)} \circ \ldots \circ \mathcal{E}_C[\![\mathrm{E}_n]\!]\beta)\,\sigma. \end{aligned}$$

We are therefore required to show that

$$\underline{eval} \circ \underline{mkap} \circ \ldots \circ \underline{mkap} \circ \underline{pushvalue}(\mathrm{I}\ \mathbf{in}\ \mathbf{N}) = \underline{call}\,\mathrm{I}\,n.$$

But

$$\begin{aligned} \acute{\sigma}_2 &= (\underline{mkap} \circ \ldots \circ \underline{mkap} \circ \underline{pushvalue}(\mathrm{I}\ \mathbf{in}\ \mathbf{N}))\,\sigma_1 \\ &= (o, \ell'_n : \phi, \psi, \gamma \oplus \gamma_{\text{ spine}}, \rho, \delta), \end{aligned}$$

where $\gamma_{\text{ spine}}$ is

$$\{\ell'_n \mapsto (\ell'_{n-1}, \ell_n)\ \mathbf{in}\ \mathbf{N}, \ldots, \ell'_1 \mapsto (\ell'_0, \ell_1)\ \mathbf{in}\ \mathbf{N}, \ell'_0 \mapsto \mathrm{I}\ \mathbf{in}\ \mathbf{N}\}.$$

Now $\underline{eval}\,\acute{\sigma}_2$ is

$$(\underline{restore} \circ \underline{exit}\,n \circ \kappa')(o, [\ell_1, \ldots, \ell_n, \ell'_n], \psi, \gamma \oplus \gamma_{\text{ spine}}, \rho, \phi : \delta),$$

so consider the evaluation of $\underline{call}\,\mathrm{I}\,n\,\sigma_1$. We can expand this to

$$(\underline{eval} \circ \underline{slide}\,n \circ \kappa')\,\sigma_1.$$

We notice that the same graph labels occupy the top n spaces of the stack in each case. Also the graphs $\acute{\gamma}_2$ and $\grave{\gamma}_2$ differ only in the assignment of $\ell'_0 \ldots \ell'_n$ labels in $\acute{\gamma}_2$.

Thus

$$\acute{\sigma}_3 = \kappa'\,\acute{\sigma}_2 = (o, [\ell, \ell_1, \ldots, \ell_n, \ell'_n], \psi, \acute{\gamma}_3, \rho, \phi : \delta),$$

where $\dot{\gamma}_3 = \dot{\gamma}_3 \oplus \gamma_{\text{spine}}$. Also

$$\dot{\sigma}_3 = \kappa' \dot{\sigma}_2 = (o, \ell : \ell_1 : \ldots : \ell_n : \ell'_n : \phi, \psi, \dot{\gamma}_3, \rho, \delta).$$

We now observe that we may expand $\underline{exit}\, n$ to $\underline{unwind} \circ \underline{pop}\, n \circ \underline{update}\, n$, so that

$$\begin{aligned} \dot{\sigma}_4 &= (\underline{pop}\, n \circ \underline{update}\, n)\, \dot{\sigma}_3 \\ &= (o, [\ell'_n], \psi, \dot{\gamma}_3 \oplus \{\ell'_n \mapsto \ell \text{ in } \mathbf{N}\}, \rho, \phi : \delta), \end{aligned}$$

and this is

$$\underline{eval}(o, \ell'_n : \phi, \psi, \dot{\gamma}_3 \oplus \{\ell'_n \mapsto \ell \text{ in } \mathbf{N}\}, \rho, \delta).$$

Notice that

$$\underline{slide}\, n\, (o, \ell : \ell_1 : \ldots : \ell_n : \ell'_n : \phi, \psi, \dot{\gamma}_3, \rho, \delta) = (o, \ell : \phi, \psi, \dot{\gamma}_3, \rho, \delta).$$

But

$$(o, \ell'_n : \phi, \psi, \dot{\gamma}_3 \oplus \{\ell'_n \mapsto \ell \text{ in } \mathbf{N}\}, \rho, \delta) \cong (o, \ell : \phi, \psi, \dot{\gamma}_3, \rho, \delta).$$

□

We may now demonstrate further improvements in the particular case of the built-in functions. These are stated in Theorem 8.11.

Theorem 8.11

For all E, E_0, E_1 *and* E_2 *in* Exp *and all* β *in* **B**:

$$\begin{aligned} \mathcal{E}_E[\![\text{null E}]\!]\beta &\cong \underline{null} \circ \mathcal{E}_E[\![\text{E}]\!]\beta \\ \mathcal{E}_E[\![\text{hd E}]\!]\beta &\cong \underline{eval} \circ \underline{hd} \circ \mathcal{E}_E[\![\text{E}]\!]\beta \\ \mathcal{E}_E[\![\text{cons } E_0\, E_1]\!]\beta &\cong \underline{cons} \circ \mathcal{E}_C[\![E_0]\!]\beta^{+1} \circ \mathcal{E}_C[\![E_1]\!]\beta \\ \mathcal{E}_E[\![\text{add } E_0\, E_1]\!]\beta &\cong \underline{add} \circ \mathcal{E}_E[\![E_0]\!]\beta^{+1} \circ \mathcal{E}_E[\![E_1]\!]\beta \\ \mathcal{E}_E[\![\text{eq } E_0\, E_1]\!]\beta &\cong \underline{eq} \circ \mathcal{E}_E[\![E_0]\!]\beta^{+1} \circ \mathcal{E}_E[\![E_1]\!]\beta \\ \mathcal{E}_E[\![\text{if } E_0\, E_1\, E_2]\!]\beta &\cong \underline{if}(\mathcal{E}_E[\![E_1]\!]\beta)(\mathcal{E}_E[\![E_2]\!]\beta) \circ \mathcal{E}_E[\![E_0]\!]\beta \end{aligned}$$

We must first prove Lemma 8.12. This states that the body of a combinator may be reduced before updating. This is analogous to the result we proved for Theorem 6.5, in Chapter 6.

Lemma 8.12

For all E *in* Exp, *all* β *in* **B**, *and all* $n > 0$

$$\mathcal{E}_R[\![\mathrm{E}]\!]\,\beta\,n \cong \underline{exit}\,n \circ \mathcal{E}_E[\![\mathrm{E}]\!]\,\beta.$$

This follows from Theorem 6.5.

Proof of Lemma 8.12

Corollary from Chapter 6, on alternative sharing mechanisms.

□

To prove the final isomorphism, we will also require Lemma 8.13.

Lemma 8.13

For all κ_T *and* κ_F, *and completely strict* κ *in* **K**,

$$\kappa \circ \underline{if}\,\kappa_T\,\kappa_F \cong \underline{if}\,(\kappa \circ \kappa_T)\,(\kappa \circ \kappa_F).$$

Notice that we need the complete strictness of the continuation κ. This condition is guaranteed because all of the instructions we have defined are completely strict; thus the composition of an arbitrary number of instructions is also completely strict.

Proof of Lemma 8.13

Cases on the state:

$\sigma = \perp$ **or** $\sigma = \underline{?}$ In these cases we observe that $\underline{if}\,\kappa_T\,\kappa_F\,\sigma = \sigma$, for all κ_T and κ_F. We also note that, for all completely strict κ; $\kappa\,\sigma = \sigma$. Thus both alternatives are equivalent to the identity function on these improper states.

$\sigma = (o, \ell : \phi, \psi, \gamma, \rho, \delta)$ We now assume that we have a proper state. There are two cases to consider, depending on the truth value on top of the stack. Notice that *if* produces an error if this is not the case, and we may appeal to the result of the previous case to demonstrate the equivalence.

$\gamma\,\ell$ | **T holds** In this case $\underline{if}\,\kappa_T\,\kappa_F\,\sigma = \kappa_T\,\sigma'$, where $\sigma' = pop\,1\,\sigma$. We are thus able to show that

$$\begin{aligned}\kappa \circ \underline{if}\,\kappa_T\,\kappa_F \circ \sigma &= (\kappa \circ \kappa_T)\sigma' \\ &= \underline{if}\,(\kappa \circ \kappa_T)\,(\kappa \circ \kappa_F)\,\sigma.\end{aligned}$$

$\gamma\,\ell$ | **T does not hold** In a similar way $\underline{if}\,\kappa_T\,\kappa_F\,\sigma$ **is** now $\kappa_F\,\sigma'$, and so

$$\begin{aligned}\kappa \circ \underline{if}\,\kappa_T\,\kappa_F \circ \sigma &= (\kappa \circ \kappa_F)\sigma' \\ &= \underline{if}\,(\kappa \circ \kappa_T)\,(\kappa \circ \kappa_F)\,\sigma.\end{aligned}$$

□

We may now prove Theorem 8.11 by cases.

Proof of Theorem 8.11

These theorems are proved using Theorem 8.10, and by observing that we may rearrange stack operations, provided the evaluation of graphs occurs in the same order.

⟦null⟧ In this case κ' is

$$\underline{null} \circ \underline{eval} \circ \underline{push}\,0.$$

We therefore observe that $\underline{call}\,⟦\text{null}⟧\,1$ is $\underline{eval} \circ \underline{slide}\,1 \circ \kappa'$. Thus

$$\begin{aligned}\underline{call}\,⟦\text{null}⟧\,1 \circ \mathcal{E}_C⟦\text{E}⟧\,\beta &= \underline{eval} \circ \underline{null} \circ \underline{eval} \circ \mathcal{E}_C⟦\text{E}⟧\,\beta \\ &= \underline{null} \circ \mathcal{E}_E⟦\text{E}⟧\,\beta.\end{aligned}$$

Notice that the second *eval* is redundant because the *null* instruction leaves its result in WHNF.

⟦hd⟧ In this case κ' is

$$\underline{hd} \circ \underline{eval} \circ \underline{push}\,0.$$

We therefore observe that $\underline{call}\,⟦\text{hd}⟧\,1$ is $\underline{eval} \circ \underline{slide}\,1 \circ \kappa'$. Thus

$$\begin{aligned}\underline{call}\,⟦\text{hd}⟧\,1 \circ \mathcal{E}_C⟦\text{E}⟧\,\beta &= \underline{eval} \circ \underline{hd} \circ \underline{eval} \circ \mathcal{E}_C⟦\text{E}⟧\,\beta \\ &= \underline{eval} \circ \underline{hd} \circ \mathcal{E}_E⟦\text{E}⟧\,\beta.\end{aligned}$$

[cons] In this case κ' is

$$\underline{cons} \circ \underline{push}\,1 \circ \underline{push}\,1.$$

We therefore observe that $\underline{call}[\![\text{cons}]\!]\,2$ is $\underline{eval} \circ \underline{slide}\,2 \circ \kappa'$. Thus

$$\begin{aligned} \underline{call}[\![\text{cons}]\!]\,2 \circ \kappa_0 \circ \kappa_1 &= \underline{eval} \circ \underline{cons} \circ \kappa_0 \circ \kappa_1 \\ &= \underline{cons} \circ \kappa_0 \circ \kappa_1, \end{aligned}$$

where $\kappa_i = \mathcal{E}_C[\![\mathrm{E}_i]\!]\beta^{+(1-i)}$. Notice that the $\underline{eval}$ instruction is redundant because the $\underline{cons}$ instruction leaves its result in WHNF.

[add] In this case κ' is

$$\underline{add} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0$$

We therefore observe that $\underline{call}[\![\text{add}]\!]\,2$ is $\underline{eval} \circ \underline{slide}\,2 \circ \kappa'$. Thus

$$\begin{aligned} & \underline{call}[\![\text{add}]\!]\,2 \circ \mathcal{E}_C[\![\mathrm{E}_0]\!]\beta^{+1} \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta \\ = \;& \underline{eval} \circ \underline{slide}\,1 \circ \underline{add} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \\ & \qquad \mathcal{E}_C[\![\mathrm{E}_0]\!]\beta^{+1} \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta \\ \cong \;& \underline{eval} \circ \underline{add} \circ \underline{eval} \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta^{+1} \circ \underline{eval} \circ \mathcal{E}_C[\![\mathrm{E}_0]\!]\beta. \end{aligned}$$

But the $\underline{eval}$ instruction immediately following the $\underline{add}$ instruction is redundant.

[eq] In this case κ' is

$$\underline{eq} \circ \underline{eval} \circ \underline{push}\,2 \circ \underline{eval} \circ \underline{push}\,0.$$

The proof is therefore identical to that of $\underline{add}$.

[if] In this case κ' is

$$\underline{if}(\underline{push}\,1)(\underline{push}\,2) \circ \underline{eval} \circ \underline{push}\,0.$$

We therefore observe that $\underline{call}[\![\text{if}]\!]\,3$ is $\underline{eval} \circ \underline{slide}\,3 \circ \kappa'$. Thus

$$\begin{aligned} & \underline{call}[\![\text{if}]\!]\,3 \circ \mathcal{E}_C[\![\mathrm{E}_0]\!]\beta^{+2} \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta^{+1} \circ \mathcal{E}_C[\![\mathrm{E}_2]\!]\beta \\ = \;& \underline{eval} \circ \underline{slide}\,2 \circ \underline{if}(\underline{push}\,1)(\underline{push}\,2) \circ \\ & \qquad \mathcal{E}_E[\![\mathrm{E}_0]\!]\beta^{+2} \circ \mathcal{E}_C[\![\mathrm{E}_1]\!]\beta^{+1} \circ \mathcal{E}_C[\![\mathrm{E}_2]\!]\beta \\ \cong \;& \underline{eval} \circ \underline{if}(\mathcal{E}_C[\![\mathrm{E}_1]\!]\beta)(\mathcal{E}_C[\![\mathrm{E}_2]\!]\beta) \circ \mathcal{E}_E[\![\mathrm{E}_0]\!]\beta. \end{aligned}$$

But the $\underline{eval}$ instruction, being completely strict, may be distributed through the $\underline{if}$ instruction by Theorem 8.13; this gives

$$\underline{if}(\mathcal{E}_E[\![\mathrm{E}_1]\!]\beta)(\mathcal{E}_E[\![\mathrm{E}_2]\!]\beta) \circ \mathcal{E}_E[\![\mathrm{E}_0]\!]\beta.$$

□

Finally, we can demonstrate that tail recursion distributes through conditional expressions. This is done in Corollary 8.14.

Corollary 8.14

For all E, E_0, E_1 *and* E_2 *in* Exp, *all* β *in* **B** *and all* $n > 0$*:*

$$\mathcal{E}_R[\![\text{if } E_0\ E_1\ E_2]\!]\beta\, n \cong \underline{if}(\mathcal{E}_R[\![E_1]\!]\beta\, n)(\mathcal{E}_R[\![E_2]\!]\beta\, n) \circ \mathcal{E}_E[\![E_0]\!]\beta.$$

This is a straightforward application of Theorem 8.11 and Lemmas 8.12 and 8.13.

Proof of Corollary 8.14

By Theorem 8.12,

$$\mathcal{E}_R[\![\text{if } E_0\ E_1\ E_2]\!]\beta\, n \cong \underline{exit}\, n \circ \mathcal{E}_E[\![\text{if } E_0\ E_1\ E_2]\!]\beta.$$

But, by Theorem 8.11,

$$\mathcal{E}_E[\![\text{if } E_0\ E_1\ E_2]\!]\beta \cong \underline{if}(\mathcal{E}_E[\![E_1]\!]\beta)(\mathcal{E}_E[\![E_2]\!]\beta) \circ \mathcal{E}_E[\![E_0]\!]\beta.$$

Now, by Theorem 8.13, we have

$$\mathcal{E}_R[\![\text{if } E_0\ E_1\ E_2]\!]\beta\, n \cong \underline{if}(\underline{exit}\, n \circ \mathcal{E}_E[\![E_1]\!]\beta)(\underline{exit}\, n \circ \mathcal{E}_E[\![E_2]\!]\beta) \circ \mathcal{E}_E[\![E_0]\!]\beta.$$

Now, by applying Theorem 8.12 again, we have

$$\mathcal{E}_R[\![\text{if } E_0\ E_1\ E_2]\!]\beta\, n \cong \underline{if}(\mathcal{E}_R[\![E_1]\!]\beta\, n)(\mathcal{E}_R[\![E_2]\!]\beta\, n) \circ \mathcal{E}_E[\![E_0]\!]\beta.$$

□

Using Johnsson's notation, we have analysed the $\mathcal{E}$ and $\mathcal{R}$-schemes and shown the required isomorphisms for these proposed compilation methods. In the next section we consider graph-isomorphic code that performs operations on a value stack, V.

$$
\begin{array}{lcl}
\underline{nullv}\sigma & = & (o, \phi, (\gamma\,\ell = nil) : \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } \sigma = (o, \ell : \phi, \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{addv}\sigma & = & (\zeta_0 \in \mathbf{Z} \wedge \zeta_1 \in \mathbf{Z}) \longrightarrow (o, \phi, \zeta : \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \zeta = \zeta_0 \mid \mathbf{Z} + \zeta_1 \mid \mathbf{Z} \\
 & & \qquad \sigma = (o, \phi, \zeta_1 : \zeta_0 : \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{eqv}\sigma & = & (\zeta_0 \in \mathbf{Z} \wedge \zeta_1 \in \mathbf{Z}) \longrightarrow (o, \phi, \tau : \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \tau = \zeta_0 \mid \mathbf{Z} = \zeta_1 \mid \mathbf{Z} \\
 & & \qquad \sigma = (o, \phi, \zeta_1 : \zeta_0 : \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{ifv}\kappa_T\,\kappa_F\,\sigma & = & (\tau \in \mathbf{T}) \longrightarrow ((\tau \mid \mathbf{T}) \longrightarrow \kappa_T, \kappa_F)(o, \phi, \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \sigma = (o, \phi, \tau : \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{mkint}\sigma & = & (\zeta \in \mathbf{Z}) \longrightarrow \underline{pushvalue}(\zeta \text{ in } \mathbf{N})(o, \phi, \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \sigma = (o, \phi, \zeta : \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{mkbool}\sigma & = & (\tau \in \mathbf{T}) \longrightarrow \underline{pushvalue}(\tau \text{ in } \mathbf{N})(o, \phi, \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \sigma = (o, \phi, \tau : \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{get}\sigma & = & (\nu \in \mathbf{Z}) \longrightarrow (o, \phi, (\nu \mid \mathbf{Z}) : \psi, \gamma, \rho, \delta) \\
 & & (\nu \in \mathbf{T}) \longrightarrow (o, \phi, (\nu \mid \mathbf{T}) : \psi, \gamma, \rho, \delta), \underline{?} \\
 & & \quad \text{where } \nu = \gamma\,\ell \\
 & & \qquad \sigma = (o, \ell : \phi, \psi, \gamma, \rho, \delta) \\
 & & \\
\underline{get2}\sigma & = & (\underline{get} \circ \underline{get})(o, \ell_0 : \ell_1 : \phi, \psi, \gamma, \rho, \delta) \\
 & & \quad \text{where } \nu = \gamma\,\ell \\
 & & \qquad \sigma = (o, \ell_1 : \ell_0 : \phi, \psi, \gamma, \rho, \delta)
\end{array}
$$

Figure 8.2: Instructions for Built-in Functions (using V)

8.3 A Stack for Basic Values

So far we have not used the third component of the state at all, this is now remedied. It is intended that this component should he a stack of integer or boolean values, and so we now provided instructions and alternative compilations for this use.

Before investigating graph-isomorphic continuations, we digress a little to

mention typing. We will presume that our language is polymorphically typed in the manner described by Milner in [1978]. We shall presume that we have a function $\mathcal{T}$ that returns the type of an expression. For most compilation purposes we are only interested in these types if they are Integer or Boolean.

Definition 8.15

If $\mathcal{T}[\![\mathrm{E}_I]\!] = \text{Integer}$ or $\mathcal{T}[\![\mathrm{E}_B]\!] = \text{Boolean}$, define $\mathcal{E}_B$ so that

$$\begin{aligned}\mathcal{E}_B[\![\mathrm{E}_I]\!]\beta &= \underline{get} \circ \mathcal{E}_E[\![\mathrm{E}_I]\!]\beta \\ \mathcal{E}_B[\![\mathrm{E}_B]\!]\beta &= \underline{get} \circ \mathcal{E}_E[\![\mathrm{E}_B]\!]\beta\end{aligned}$$

We would now like to show that we may use the alternative continnations produced by the new compilation rule $\mathcal{E}_B$. The selection we consider is given in Theorem 8.16.

Theorem 8.16

For all well-typed programs, the following pairs of continuations are equivalent:

$$\begin{aligned}&\mathcal{E}_B[\![\mathrm{null\ E}]\!]\beta && \cong && \underline{nullv} \circ \mathcal{E}_E[\![\mathrm{E}]\!]\beta \\ &\mathcal{E}_B[\![\mathrm{add\ E_0\ E_1}]\!]\beta && \cong && \underline{addv} \circ \mathcal{E}_B[\![\mathrm{E_1}]\!]\beta \circ \mathcal{E}_B[\![\mathrm{E_0}]\!]\beta \\ &\mathcal{E}_B[\![\mathrm{eq\ E_0\ E_1}]\!]\beta && \cong && \underline{eqv} \circ \mathcal{E}_B[\![\mathrm{E_1}]\!]\beta \circ \mathcal{E}_B[\![\mathrm{E_0}]\!]\beta \\ &\mathcal{E}_R[\![\mathrm{if\ E_0\ E_1\ E_2}]\!]\beta\, n && \cong && \underline{ifv}(\mathcal{E}_R[\![\mathrm{E_1}]\!]\beta\, n)(\mathcal{E}_R[\![\mathrm{E_2}]\!]\beta\, n) \circ \mathcal{E}_B[\![\mathrm{E_0}]\!]\beta \\ &\mathcal{E}_E[\![\mathrm{if\ E_0\ E_1\ E_2}]\!]\beta && \cong && \underline{ifv}(\mathcal{E}_E[\![\mathrm{E_1}]\!]\beta)(\mathcal{E}_E[\![\mathrm{E_2}]\!]\beta) \circ \mathcal{E}_B[\![\mathrm{E_0}]\!]\beta \\ &\mathcal{E}_B[\![\mathrm{if\ E_0\ E_1\ E_2}]\!]\beta && \cong && \underline{ifv}(\mathcal{E}_B[\![\mathrm{E_1}]\!]\beta)(\mathcal{E}_B[\![\mathrm{E_2}]\!]\beta) \circ \mathcal{E}_B[\![\mathrm{E_0}]\!]\beta\end{aligned}$$

It is interesting to notice that the "obvious" isomorphism

$$\underline{mkint} \circ \mathcal{E}_B[\![\mathrm{E}_I]\!]\beta \cong \mathcal{E}_E[\![\mathrm{E}_I]\!]\beta,$$

is not, in general true. This is because the *mkint* instruction has created a copy of a node, and hence the structure of the two graphs is no longer the same. We note that this occurs only when the original node was shared, and so we are able to prove a result of sufficient power, by considering only the unshared case.

Lemma 8.17

$$\begin{aligned}\underline{get} \circ \underline{mkint} \circ \underline{addv} &\cong \underline{addv} \\ \underline{get} \circ \underline{mkbool} \circ \underline{eqv} &\cong \underline{eqv} \\ \underline{get} \circ \underline{mkbool} \circ \underline{nullv} &\cong \underline{nullv}\end{aligned}$$

Notice that in all cases the built-in instruction leaves an unshared node on top of the stack. We use this property to prove Lemma 8.17.

Proof of Lemma 8.17

Suppose $addv\,\sigma = (o, \phi, \zeta : \psi, \gamma, \rho, \delta)$. Then

$$(\underline{get} \circ \underline{mkint})\,(o, \phi, \zeta : \psi, \gamma, \rho, \delta) = \underline{get}(o, \ell : \phi, \psi, \gamma \oplus \{\ell \mapsto \zeta \text{ in } \mathbf{N}\}, \rho, \delta).$$

But this is $(o, \phi, \zeta : \psi, \gamma \oplus \{\ell \mapsto \zeta \text{ in } \mathbf{N}\}, \rho, \delta)$. However,

$$(o, \phi, \zeta : \psi, \gamma \oplus \{\ell \mapsto \zeta \text{ in } \mathbf{N}\}, \rho, \delta) \cong (o, \phi, \zeta : \psi, \gamma, \rho, \delta),$$

because there is no longer a reference to the new node labelled by ℓ.

By a similar argument, we may show that

$$(\underline{get} \circ \underline{mkbool})\,(o, \phi, \tau : \psi, \gamma, \rho, \delta) = (o, \phi, \tau : \psi, \gamma \oplus \{\ell \mapsto \tau \text{ in } \mathbf{N}\}, \rho, \delta).$$

Again, there are no longer references to the new nodes labelled ℓ, and this allows us to deduce the last two isomorphisms.

□

We will also need to demonstrate the relationship between the original built-in instructions and their V-stack equivalents. This is done in Lemma 8.18.

Lemma 8.18

$$\begin{array}{lcl} \underline{mkbool} \circ \underline{nullv} & = & \underline{null} \\ \underline{mkint} \circ \underline{addv} \circ \underline{get2} & = & \underline{add} \\ \underline{mkbool} \circ \underline{eqv} \circ \underline{get2} & = & \underline{eq} \\ \underline{ifv}\,\kappa_T\,\kappa_F \circ \underline{get} & = & \underline{if}\,\kappa_T\,\kappa_F \end{array}$$

The proof is by straightforward substitution.

Proof of Lemma 8.18

By direct substitution for the instructions *mkint*, *mkbool*, *addv*, *eqv*, *nullv*, *ifv*, *get* and *get2*.

□

Finally, we show the relationship between the sequencing of two graph evaluations, under the $\mathcal{E}_E$ and $\mathcal{E}_B$ compilation schemes.

Lemma 8.19

For all E_0 *and* E_1 *in* Exp *and all* β *in* **B**:

$$\underline{get2} \circ \mathcal{E}_E[\![\mathrm{E}_1]\!]\beta^{+1} \circ \mathcal{E}_E[\![\mathrm{E}_0]\!]\beta = \mathcal{E}_B[\![\mathrm{E}_1]\!]\beta \circ \mathcal{E}_B[\![\mathrm{E}_0]\!]\beta.$$

Again, the proof is direct substitution.

Proof of Lemma 8.19

Apart from checking the stack order, this proof is simple. Suppose that

$$\mathcal{E}_E[\![\mathrm{E}_0]\!]\beta\,\sigma = (o, \ell_0 : \phi, \psi, \gamma_0, \rho, \delta).$$

Then

$$\mathcal{E}_E[\![\mathrm{E}_1]\!]\beta^{+1}\,(o, \ell_0 : \phi, \psi, \gamma_0, \rho, \delta) = (o, \ell_1 : \ell_0 : \phi, \psi, \gamma_1, \rho, \delta).$$

However,

$$(\underline{get} \circ \mathcal{E}_E[\![\mathrm{E}_1]\!]\beta \circ \underline{get})\,(o, \ell_0 : \phi, \psi, \gamma_0, \rho, \delta) = (o, \phi, \zeta_1 : \zeta_0 : \psi, \gamma_1, \rho, \delta),$$

where $\zeta_i = \gamma_1\,\ell_i \mid \mathbf{Z}$. But this is precisely

$$\underline{get2}(o, \ell_1 : \ell_0 : \phi, \psi, \gamma_1, \rho, \delta).$$

□

We may now prove Theorem 8.16.

Proof of Theorem 8.16

We prove this by cases:

$\mathcal{E}_B[\![\text{null E}]\!]\beta$ Firstly,

$$\mathcal{E}_B[\![\text{null E}]\!]\beta = \underline{get} \circ \mathcal{E}_E[\![\text{null E}]\!]\beta.$$

But, by Theorem 8.11, we have

$$\mathcal{E}_E[\![\text{null E}]\!]\beta = \underline{null} \circ \mathcal{E}_E[\![\text{E}]\!]\beta.$$

Also $\underline{null} = \underline{mkbool} \circ \underline{nullv}$ and hence, by Lemma 8.17, $\underline{get} \circ \underline{null} \cong \underline{nullv}$. Thus

$$\mathcal{E}_B[\![\text{null E}]\!]\beta \cong \underline{nullv} \circ \mathcal{E}_E[\![\text{E}]\!]\beta.$$

$\mathcal{E}_B[\![\text{add } \text{E}_0\ \text{E}_1]\!]\beta$ Firstly,

$$\mathcal{E}_B[\![\text{add } \text{E}_0\ \text{E}_1]\!]\beta = \underline{get} \circ \mathcal{E}_E[\![\text{add } \text{E}_0\ \text{E}_1]\!]\beta.$$

But, by Theorem 8.11, we have

$$\mathcal{E}_E[\![\text{add } \text{E}_0\ \text{E}_1]\!]\beta = \underline{add} \circ \mathcal{E}_E[\![\text{E}_1]\!]\beta^{+1} \circ \mathcal{E}_E[\![\text{E}_0]\!]\beta.$$

Also, by Lemmas 8.18 and lm:8-3, $\underline{add} = \underline{mkint} \circ \underline{addv} \circ \underline{get2}$ and $\underline{get} \circ \underline{mkint} \circ \underline{addv} \cong \underline{addv}$. Hence

$$\mathcal{E}_B[\![\text{add } \text{E}_0\ \text{E}_1]\!]\beta \cong \underline{addv} \circ \underline{get2} \circ \mathcal{E}_E[\![\text{E}_1]\!]\beta^{+1} \circ \mathcal{E}_E[\![\text{E}_0]\!]\beta.$$

But, by Lemma 8.19,

$$\underline{get2} \circ \mathcal{E}_E[\![\text{E}_1]\!]\beta^{+1} \circ \mathcal{E}_E[\![\text{E}_0]\!]\beta = \mathcal{E}_B[\![\text{E}_1]\!]\beta \circ \mathcal{E}_B[\![\text{E}_0]\!]\beta.$$

from which we obtain the result.

$\mathcal{E}_B[\![\text{eq } \text{E}_0\ \text{E}_1]\!]\beta$ This is proved in the same way as the previous case.

$\mathcal{E}_R[\![\text{if } \text{E}_0\ \text{E}_1\ \text{E}_2]\!]\beta\, n$ This case follows immediately from Theorem 8.14 and Lemma 8.18.

$\mathcal{E}_E[\![\text{if } \text{E}_0\ \text{E}_1\ \text{E}_2]\!]\beta$ This case follows immediately from Theorem 8.11 and Lemma 8.18.

$\mathcal{E}_B[\![\text{if } \text{E}_0\ \text{E}_1\ \text{E}_2]\!]\beta$ This case follows immediately from the previous case and Lemma 8.13, which allows us to propagate the $\underline{get}$ into both branches of the conditional.

□

This concludes our brief survey of some of the code improvement techniques used in the G-machine.

8.4 Related Work

Most of the material in this chapter can be found in Sections 2 and 3 of [Lester 87], where graph-isomorphism and its uses are described. In [1987], Augustsson provides source-level transformations for his G-machine and its related denotational semantics. His proofs are made using denotational semantics.

In a paper on the Scheme 311 compiler [1984], Clinger performs a similar set of transformations, but as he remarks, transformations such as

$$[\![\texttt{(lambda (x) x) 3}]\!] \Rightarrow [\![\texttt{3}]\!]$$

may not be valid. The storage requirements may change. This is precisely the problem graph-isomorphism is introduced to solve. We note, however, that Clinger is correct when he states that store-semantics congruences are far messier than the corresponding direct-semantics congruences. The direct denotational semantics for Scheme with which he works is more complex than the one presented in this thesis, but his operational semantics is of similar complexity to that which we describe. This provides yet another reason why we would like to work with the denotational semantics rather than the operational semantics if at all possible.

The optimizations we discuss here were first proposed by Johnsson in [1983]. He proposes separation of the two functions of the dump in his thesis [Johnsson 87].

8.5 Conclusion

The useful operational equivalence of graph-isomorphism is defined in Section 8.1. It is used in Section 8.2, where we have seen how we may avoid the construction of unnecessary graph. This is accomplished by reducing built-in functions directly, if their result will necessarily be required. A more accurate strictness abstract interpretation could be used to improve the quality of the generated code. In Section 8.3 we extend this idea, so that intermediate results of built-in functions are retained in V instead of being written out to the heap only to be retrieved later. Both of these ideas were originally described in [Johnsson 83].

Chapter 9

Conclusion and Further Work

To conclude, we briefly summarize the main results presented in this thesis.

9.1 Results

Underlying this thesis is the desire to present an implementation of a lazy functional language, and to show that this implementation satisfies the declarative properties that we expect.

We have chosen to use graph reduction as the basis of our implementation. This is because both TIM [Fairburn and Wray 87] and the G-machine [Johnsson 83], two of the most efficient implementations of lazy functional languages, are based on the graph reduction model. We have presented a formal model of graph reduction. This model was intended to accurately represent the method of implementation used by most current graph reduction implementations of functional languages. For this reason, error handling was included in the definition of the model. The model used indirection nodes to maintain sharing, and because of this the model was naturally tail-recursive.

The model of graph reduction was then shown to be congruent to the standard denotational semantics, which provides a definition of the declarative properties of a lazy functional program. Although both models define the same language, the operational model has the additional property that its operational behaviour is specified. As a result of the existence of the congruence, we are able to state that the graph reduction model is correct with respect to the denotational semantics. The congruence proof follows

the usual pattern of such proofs, by establishing that each semantics approximates the other. The novelty of the congruence proof presented in Chapters 3 and 4 is that the language we present has lazy rather than the strict function application semantics presented by Stoy in [1981].

The first application of the congruence is to discover the condition under which the evaluation order may be re-arranged. This condition is complete strictness which is a stronger condition than that typically determined by strictness analyses for the λ-calculus.

It was then discovered that the language was not confluent because of a design decision for the implementation of addition and equality. Techniques to restore the property of confluence to the language are discussed in Chapter 5.

We investigated alternative sharing mechanisms in Chapter 6. The conclusion was reached that copying an evaluated graph preserved sharing, and thus is a legitimate strategy. To prove this we had to prove Theorem 6.2, which concerns cycles in the spine of a graph. It was shown that a state with a spine cycle was necessarily non-terminating. Implicit in the original operational semantics was the existence of two fixpoints. The first or outer one controls the sequence of reduction steps, and the inner one is part of the specification of the spine unwinding process. We observe that either may cause non-termination. There is an important difference between them however. A spine cycle causes a detectable $\perp$, because the graph is finite. A non-termination occurring because we reach no final state remains undetectable because it is equivalent to the halting problem.

A new sharing scheme was proposed which combines the indirection and copying forms of sharing, so that chains of indirection nodes do not accumulate and also we do not create new stack frames to evaluate partial applications to weak head normal form.

In Chapter 7 we demonstrated that the operational semantics may be represented by a small set of combinators. These combinators correspond to the instruction set of a G-machine abstract machine. We are therefore able to conclude that the G-machine also provides a correct implementation with respect to the language specified by the denotational semantics.

The G-machine includes some interesting optimizations. After we define a modified form of graph isomorphism, we are able to demonstrate that some of the alternative code sequences generated by the optimizations are isomorphic to the original code sequences. This is done in Chapter 8.

This concludes the work covered in this thesis and we now consider work which could usefully be done to extend it.

9.2 Further Work

Although this thesis has answered some of the basic questions arising from the use of graph reduction to implement functional programming languages, there are a number of extensions and open problems that remain.

The langnage used in this thesis is distinctly minimal. Many modern languages have additional features, such as local definitions, list comprehensions and pattern matching. These features could be compiled into the language we have used. Source to source translation is not usually used in denotational semantics, as the meaning of a program fragment is then not easily discovered. This problem is carried over into the programmers model for the language, making programs more difficult to reason about. When we consider the operational semantics of the new language features, we see that it is usual to provide more efficient implementations than those produced by the simple translations. It would therefore be interesting to know whether these extensions to the language lead to complications when a congruence proof is attempted.

The transformational development of the G-machine, provided in Chapter 7, could usefully be adapted to derive other abstract machine models, such as TIM [Fairburn and Wray 87]. Unfortunately it is not possible to investigate the relative efficiency of these two implementations, other than statistically. This serves as a warning that we should not expect to be able to derive the "best" abstract machine from the operational semantics of graph reduction. One attempt to provide a more efficient model is the Spineless G-Machine [Burn *et al.* 88] which includes a transformational derivation of TIM. Another is Argo's G-TIM [Argo 88]. Argo has investigated the relative merits of the G-machine and TIM, and then incorporated the best features of each in the G-TIM. It would still be interesting to know of other useful implementations to be found in the space of possible derivations.

As an alternative to the operational semantics provided in Chapters 2 and 4, it would be interesting to consider a parallel operational semantics. There is an important difference between the parallel and sequential models of graph reduction. The sequential algorithm has no real choice about which redex to reduce at each step. With a parallel implementation, not only is there a choice of redexes, but this choice can be non-deterministic. There are therefore many parallel operational models for graph reduction, whilst there is essentially only one sequential model. It follows from this that a transformational development, like that of Chapter 7, is not likely to generate all of the interesting parallel models.

Bibliography

[1988] G. Argo.
The G-TIM: A refiued three instruction machine.
Technical report, Glasgow University, 1988.
To appear.

[1984] L. Augustsson.
A compiler for lazy ML.
In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, Texas, August 1984.

[1985] L. Augustsson.
Compiling pattern matching.
In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, Nancy, France, September 1985.

[1987] L. Augustsson.
Compiling Lazy Functional Languages, Part II.
Doctoral thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[1981] H.P. Barendregt.
The Lambda Calculus, volume 105 of *Studies in Logic and the Foundations of Mathematics.*
Elsevier Science Publishers B.V., P.O. Box 1991, 1000 BZ Amsterdam, The Netherlands, 1981.
ISBN 0-444-87508-5.

[1988] R.J. Bird and P.L. Wadler.
An Introduction to Functional Programming.
Prentice Hall Series in Computer Science. Prentice Hall International (UK) Ltd., Hemel Hempstead, Hertfordshire, England,

1988.
ISBN 0-13-484189-1.

[1975] W.H. Burge.
Recursive Programming Techniques.
The Systems Programming Series. Addison-Wesley Publishing Company, Inc., 1975.
ISBN 0-201-14450-6.

[1986] G.L. Burn, C.L. Hankin, and S. Abramsky.
Strictness analysis of higher-order functions.
Science of Computer Programming, 7:249–278, November 1986.

[1988] G.L. Burn, S.L. Peyton Jones, and J.D. Robson.
The spineless G-machine.
In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[1969] R.M. Burstall.
Proving properties of programs by structural induction.
The Computer Journal, 12(1):41–48, February 1969.

[1936] A. Church.
An unsolvable problem in elementary number theory.
American Journal of Mathematics, 58:345–363, 1936.

[1941] A. Church.
The Calculi of Lambda-Conversion.
Princeton University Press, Princeton, N.J., 1941.

[1980] T.J.W. Clarke, P.J.S. Gladstone, C.D. Maclean, and A.C. Norman.
SKIM - the SKI reduction machine.
In *Proceedings of the ACM Lisp Conference*, Stanford, 1980.

[1984] W. Clinger.
The Scheme 311 Compiler. An exercise in denotational semantics.
In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 356–364, Austin, Texas, August 1984.

[1977] P. Cousot and R. Cousot.
Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points.
In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.

[1986] P.-L. Curien.
Categorical Combinators, Sequential Algorithms And Functional Programming.
Research Notes in Theoretical Computer Science series. Pitman Publishing Limited, London, 1986.
ISBN 0-470-20290-4.

[1930] H.B. Curry.
Grundlagen der kombinatorischen logik.
American Journal of Mathamatics, 52:509–536 and 789–834, 1930.

[1958] H.B. Curry and R. Feys.
Combinatory Logic, volume 1.
North Holland Publishing Company, Amsterdam, 1958.

[1976] P.J. Downey and R. Sethi.
Correct computation rules for recursive languages.
SIAM Journal on Computing, 5(3):378–401, September 1976.

[1987] J. Fairburn and S. Wray.
TIM: A simple, lazy abstract machine to execute supercombinators.
In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 34–45. Springer-Verlag, September 1987.

[1987] B. Goldberg.
Detecting sharing of partial applications in functional languages.
In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 408–425. Springer-Verlag, September 1987.

[1973] M.J.C. Gordon.
Models of pure LISP (a worked example in semantics).
Experimental Programming Reports 31, Department of Machine Intelligence, University of Edinburgh, 1973.

[1979] M.J. Gordon, R. Milner, and C.P. Wadsworth.
Edinburgh LCF, volume 78 of *Lecture Notes In Computer Science.*
Springer Verlag, Berlin, 1979.

[1969] F. Harary.
Graph Theory.
Addison-Wesley Publishing Co., Reading, Massachusetts, 1969.

[1980] P. Henderson.

Functional Programming: Application and Implementation.
Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd., London, 1980.
ISBN 0-13-331579-7.

[1976] P. Henderson and J.M. Morris.
A lazy evaluator.
In *Proceedings of the Third POPL Symposium*, pages 95–103, Atlanta Georgia, January 1976.

[1982] P. Henderson, G.A. Jones, and S.B. Jones.
The lispkit manual.
Technical Monograph PRG-32, Oxford University Computing Laboratory, Programming Research Group, 1982.

[1982a] R.J.M. Hughes.
Graph reduction with super-combinators.
Technical Monograph PRG-28, Oxford University Computing Laboratory, Programming Research Group, June 1982.

[1982b] R.J.M. Hughes.
Super-combinators: a new implementation method for applicative languages.
In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Languages*, pages 1–10, August 1982.

[1983] R.J.M. Hughes.
The Design and Implementation of Programming languages.
Doctoral thesis, Oxford University Computing Laboratory, Programming Research Group, July 1983.
Also published as Technical Monograph PRG-40.

[1985] R.J.M. Hughes.
Lazy memo-functions.
In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture Conference*, pages 129–146, Nancy, France, September 1985.

[1983] T. Johnsson.
The G-machine. An abstract machine for graph reduction.
In *Declarative Programming Workshop*, pages 1–20, University College London, April 1983.

[1984] T. Johnsson.

Efficient compilation of lazy evaluation.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, Canada, June 1984.

[1985] T. Johnsson.
Lambda lifting: Transforming programs to recursive equations.
In J.-P. Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, Nancy, France, September 1985.

[1987] T. Johnsson.
Compiling Lazy Functional Languages.
Doctoral thesis, Chalmers Tekniska Högskola, Göteborg, Sweden, 1987.

[1986] M.B. Josephs.
Functional Programming with Side-effects.
Doctoral thesis, Oxford University Computing Laboratory, Programming Research Group, 1986.
Also published as Technical Monograph PRG-55.

[1936] S.C. Kleene.
General recursive functions of natural numbers.
Mathematical Annals, 112:727–742, 1936.

[1950] S.C. Kleene.
Introduction to Metamathematics.
Von Nostrand, Princeton, 1950.

[1980] J.W. Klop.
Combinatory Reduction Systems, volume 127 of *Mathematical Centre Tracts.*
Mathematischen Centrum, 413 Kruislaan, Amsterdam, 1980.
ISBN 90-6190-200-5.

[1964] P.J. Landin.
The mechanical evaluation of expressions.
Computer Journal, 6(4):308–320, January 1964.

[1985] D.R. Lester.
The correctness of a g-machine compiler.
Transfer dissertation, Programming Research Group, Oxford, December 1985.

[1987] D. Lester.

The G-machine as a representation of stack semantics.
In G. Kahn, editor, *Proceedings of the Functional Programming Languages and Computer Architecture Conference*, pages 46–59. Springer-Verlag, September 1987.

[1973] Z. Manna, S. Ness, and J. Vuillemin.
Inductive methods for proving properties of programs.
Communications of the ACM, 16(8):491–502, August 1973.

[1962] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M. Levin.
LISP 1.5 Programmers Manual.
MIT Press, Cambridge, Massachusetts, 1962.

[1985] S.R. de Lemos Meira.
On the Efficiency of Applicative Algorithms.
Doctoral thesis, The University of Kent at Canterbury, March 1985.

[1974] R.E. Milne.
The Formal Semantics of Computer Languages and Their Implementation.
Doctoral thesis, University of Cambridge, 1974.

[1976] R.E. Milne and C. Strachey.
A Theory of Programming Language Semantics.
Chapman and Hall, London, 1976.

[1977] R. Milner.
Fully abstract models of typed lambda-calculi.
Theoretical Computer Science, 4(1):1–23, February 1977.

[1978] R. Milner.
A theory of type polymorphism in programming.
Journal of Computer and System Science, 17(3):348–375, 1978.

[1984] K. Mulmuley.
A semantic characterization of full abstraction.
Technical report, Carnegie-Mellon University, 1984.

[1986] K. Mulmuley.
Fully abstract submodels of typed lambda calculus.
Journal of Computer and System Sciences, 33:2–46, 1986.

[1981a] A. Mycroft.
Abstract Interpretation and Optimising Transformations for Applicative Programs.

Doctoral thesis, University of Edinburgh, Department of Computer Science, December 1981.
Also published as CST-15-81.

[1981b] A. Mycroft.
The theory and practice of transforming call-by-need into call-by-value.
Internal Report CSR-88-81, University of Edinburgh, Department of Computer Science, July 1981.

[1984] A. Mycroft.
Polymorphic type schemes and recursive definitions.
In *Proceedings of the International Symposium on Programming*, pages 217–239, Toulouse, France, 1984. Springer Verlag.
LNCS no 167.

[1988] C.-H. Luke Ong.
The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.
Doctoral thesis, University of London, 1988.
To appear.

[1987] S.L. Peyton Jones.
The Implementation of Functional Programming Languages.
Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd, London, 1987.
ISBN 0-13-453333-x.

[1977] G. Plotkin.
LCF considered as a programming language.
Theoretical Computer Science, 5(3):223–256, 1977.

[1974] J.C. Reynolds.
On the relation between direct and continuation semantics.
In *Proceedings of the Second Colloquium on Automata, Languages and Programming*, pages 141–156, Saarbrucken, 1974. Springer-Verlag.

[1986] D.A. Schmidt.
Denotational Semantics.
Allyn and Bacon, Inc., 7 Wells Avenue, Newton, Massachusetts, 1986.
ISBN 0-205-08974-7.

[1924] M. Schönfinkel.
Über die bausteine der mathematischen logik.
Mathamatische Annalen, 92:305–316, 1924.

[1970] D.S. Scott.
Outline of mathematical theory of computation.
Technical Monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group, 1970.

[1973] D.S. Scott.
Models for various type-free calculi.
In P. Suppes, L. Henkin, A. Joja, and G.C. Moisil, editors, *Logic, Methodology and Philosophy of Science*, pages 157–187. North-Holland, Amsterdam, 1973.

[1976] D.S. Scott.
Data types as lattices.
SIAM Journal on Computing, 5(3):522–587, September 1976.

[1981] D.S. Scott.
Lectures on a mathematical theory of computation.
Technical Monograph PRG-19, Oxford University Computing Laboratory, Programming Research Group, 1981.

[1977] J.E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
The MIT Press Series in Computer Science. MIT Press, Cambridge, Massachusetts, 1977.

[1981] J.E. Stoy.
The congruence of two programming language definitions.
Theoretical Computer Science, 13(2):151–174, February 1981.

[1982] J.E. Stoy.
Some mathematical aspects of functional programming.
In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications: An Advanced Course*, pages 217–252. Cambridge University Press, Cambridge, England, 1982.
ISBN 0-521-24503-6.

[1984] W.R. Stoye, T.J.W. Clarke, and A.C. Norman.
Some practical methods for rapid combinator reduction.

In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 159–166, Austin, Texas, 1984.

[1966] C. Strachey.
Towards a formal semantics.
In T.B. Steel, editor, *Formal Language Description Languages for Computer Programming*. North-Holland, Amsterdam, Netherlands, 1966.

[1983] R.D. Tennent.
Principles of Programming Languages.
Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK) Ltd., London, 1983.
ISBN 0-13-709873-1.

[1976] D.A. Turner.
SASL language manual.
Technical Report CS/75/1, Department of Computational Science, University of St. Andrews, 1976.

[1979a] D.A. Turner.
Another algorithm for bracket abstraction.
Journal of Symbolic Logic, 44(2), June 1979.

[1979b] D.A. Turner.
A new implementation technique for applicative languages.
Software Practice and Experience, 9(1):31–49, January 1979.

[1987] P.L. Wadler.
Strictness analysis on non-flat domains (by abstract interpretation over finite domains).
In S. Abramsky and C.L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12, pages 266–275. Ellis Horwood Ltd., Chichester, West Sussex, England, 1987.
ISBN 0-7458-0109-9.

[1971] C.P. Wadsworth.
Semantics and Pragmatics of The Lambda Calculus.
Doctoral thesis, University of Oxford, 1971.

[1976] C.P. Wadsworth.
The relation between computational and denotational properties for Scott's D_∞-models of the lambda-calculus.
SIAM Journal on Computing, 5(3):488–521, September 1976.

[1982] M. Wand.
Deriving target code as a representation of continuation semantics.
ACM Transactions on Programming Languages and Systems, 4(3):496–517, July 1982.