CONTROL OPERATORS: ISSUES OF EXPRESSIBILITY

by

DANIEL B. KEITH

A DIRECTED RESEARCH PROJECT REPORT

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December, 2008

Abstract

It has been shown that a language with the delimited control operators `shift` and `reset` can be translated into a language containing only an undelimited control operator such as `callcc` and a single mutable storage cell. We study this translation and a related translation from a language containing `callcc` into one containing the `control` and `prompt` operators. We show that the translation does not faithfully preserve other control effects that may be present. We describe new versions of these translations that address these problems. We provide a background on control operators and some of the formalisms used to describe the behavior and translation of these operators.

TABLE OF CONTENTS

CHAPTER I

Introduction

For the past few months, I have worked with my research advisor, Professor Zena Ariola, on applying formal programming language techniques to understand how various control operators behave and misbehave when translated or implemented in a target language. The questions that our work addresses are:

> Are there systematic errors in the translation of control operators that result in the translation not being faithful in the presence of other effects?
>
> Can we improve the translation to avoid these errors?

This report details my Directed Research Project on these questions, which we answer in the affirmative. To anticipate a broad audience, I will include some background on the terminology, notation, and techniques used in the formal study of control operators and programming languages.

For our purposes, the term *language* informally refers to a set (usually infinite) of possible well-formed programs, and to the means for evaluating, executing or otherwise interpreting a well-formed program. This definition applies to conventional programming languages such as C++, Java, and ML as well as to the synthetic programming languages studied by programming language researchers and discussed in this report.

A programming language may be formally characterized by providing its syntax and one or more semantics. The *syntax* for a particular language is a set of rules or other means of defining the well-formed (syntactically valid) sentences in the language. A *semantics* for a language is a means of mapping program phrases into some value or meaning. In this report, we will be primarily concerned with operational semantics, which emphasizes a view of program execution as a series of transitions, ultimately leading to some value or final result. For completeness and historical context, we will introduce denotational semantics, which emphasizes a view of a program text as having a meaning or *denotation* in some *model domain*.

Our research considered a family of functional programming languages, each member of which extends a base language L_0 (an untyped, call-by-value, lambda calculus). We looked at some of the well-known translations between members of this family, and developed some examples wherein these translations were not faithful to our expectations. Further analysis revealed a common oversight in these translations and permitted us to craft improved versions of these translations. Specifically, the implicit by-value conversion in a call-by-value language can affect the order of effects and inattention to this reordering can produce erroneous translations. The purpose of this report is to provide some background on the formal techniques used to describe these examples, to analyze their shortcomings, and to provide improved implementations.

1.1 The Language L_{+-}

For the purposes of this Introduction, we will consider an example language of arithmetic expressions which will be called L_{+-}, and will describe it using the formal methods that we will expand upon later in this report.

The intent behind our example language L_{+-} is to allow the expression and evaluation of simple addition/subtraction expressions using signed integers. Program phrases in this language will consist of arithmetic expressions that contain only integers (positive or negative), the infix operators + and −, and parentheses `()` for grouping. For example, the following are some examples of L_{+-} phrases, as well as some strings that are not:

Well-Formed ($\in L_{+-}$)	Not Well-Formed ($\notin L_{+-}$)
`1 + 2`	`--1`
`1 - ( -1 + -1 )`	$1 + ()$

1.1.1 Concrete Syntax

We can unambiguously specify the set of all *well-formed* L_{+-} programs by providing a *concrete syntax* in BNF (Backus-Naur Form), as illustrated in Figure 1.1.

$\langle digit \rangle$	$::=$	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`	Digits
$\langle num \rangle$	$::=$	$\langle digit \rangle$ |	Numbers
		$\langle num \rangle \langle digit \rangle$	
$\langle snum \rangle$	$::=$	$\langle num \rangle$ |	Signed Numbers
		+ $\langle num \rangle$ |	
		- $\langle num \rangle$	
$\langle expr \rangle$	$::=$	$\langle snum \rangle$ |	Expressions
		$\langle expr \rangle$ + $\langle expr \rangle$ |	
		$\langle expr \rangle$ - $\langle expr \rangle$ |	
		($\langle expr \rangle$)	
$\langle prog \rangle$	$::=$	$\langle expr \rangle$	

FIGURE 1.1: These rules define the concrete syntax of a simple arithmetic language L_{+-}.

Most formal programming analysis operates upon an *abstract syntax* which is even further removed from the character set and physical details of the source text. An abstract syntax for a language will define a set of syntax domains, as well as a set of rules that indicate how a particular syntax phrase may be built from smaller phrases. Each syntax domain corresponds to a particular *role* in the syntax, similar to a *part of speech* in a natural language. For our purposes, the syntactic values that populate these syntactic domains are best understood as *abstract syntax trees*. However, we will usually write these trees in a linear form using parenthesization to indicate the structure of the tree. For example, when we write the concrete syntax phrase $1 + (2 - 3)$ as the abstract $plus(1, minus(2, 3))$, we are actually representing a tree with $plus$ at the root, $minus$ as an inner node, and 1, 2, and 3 as terminal nodes.

The issues of how characters are mapped into numerals, how parentheses are used to group subexpressions, infix and postfix distinctions, and syntactic ambiguities are all ignored when using abstract syntax to describe a language. It is assumed that there exists a function to map from a concrete syntax phrase into an abstract syntax structure; this function is typically implemented as a parser, which takes as input a phrase in the concrete syntax of L_{+-} and returns a phrase in the abstract syntax of L_{+-}.

Figure 1.2 presents the abstract syntax description of L_{+-}. Notice how the abstract syntax emphasizes the compositional structure of the syntax, that an $\langle expr \rangle$ is composed of smaller syntactic elements. As is conventional in the literature, we define and use a representative *metavariable* for each syntax domain, to make the rules and subsequent reasoning more concise. In our L_{+-} syntax, we have two syntax domains: Num and $Expr$. We use the symbol N as a metavariable that represents any element of Num; similarly, E represents any element of $Expr$. We can then use these representative symbols in our syntactic and semantic descriptions. For example, $E ::= plus(E, E)$ is easier to read than the corresponding set-based description, $Expr ::= plus(Expr, Expr)$. It is important to remember that E is a metavariable that represents a particular element of $Expr$; $Expr$ is the set of all syntactic objects that are generated by the $Expr$-rules.

The abstract syntax of a language forms a multi-sorted algebra [22], where elements are either atomic (as in the case of N) or composite (as in the case of $plus(E, E)$). This algebra is independent of any particular *realization* and can be used as a means to prove properties of the syntactic structure. More importantly, this algebra can be used to easily and concisely express semantic functions, both operational and denotational.

As long as we keep in mind that the domain of discourse is abstract syntax, then we can choose to use a particular realization of this syntax in our notation; this realization is usually very similar to the concrete syntax. So we can write $1 + 2$ and know that we are referring to the abstract syntax tree $plus(number(1), number(2))$ without ambiguity. This convention simplifies the notation, while still allowing us to reason about abstract syntax. In the remainder of this document, we will usually use a notation that resembles concrete syntax for convenience.

Syntax Domains

N	$\in$	*Num* (Integers)
E	$\in$	*Expr* (Expressions)

Syntax Constructors

		Abstract	*Concrete*
N	$::=$	$number(n)$	n
E	$::=$	$value(N)$ |	N
		$plus(E, E)$ |	$E + E$
		$minus(E, E)$	$E - E$

FIGURE 1.2: The abstract syntax describes one or more syntax domains, each of which is a set of abstract syntax phrases. For each domain, there are generators to build elements of the domain.

1.1.2 Denotational Semantics

A semantics for a given language L is a way to assign a meaning to an arbitrary program $P \in L$. Denotational semantics is useful in programming language theory because it provides a mapping between a language and a model domain of mathematical functions and values. This model domain can then be used as the basis of formal reasoning about the programs in L, their capabilities and limitations.

A typical denotational semantics provides a semantic valuation function which can be applied to a program $P \in L$ to produce a value or function in the model domain. The evaluation function is usually indicated with the $[\![]\!]$ delimiters. For example, if we consider our program $1+2$ in our L_{+-} language, then $[\![1+2]\!] = 3$. Note that the expression $1+2$ is in the domain of abstract syntax, whereas the 3 is in the model domain for this semantics, which is $\mathbb{N}$. Some authors will use a notation or text style to distinguish between the $3 \in \mathbb{N}$ and the $3 \in N$; for example, the above might be written as $[\![\overline{1} + \overline{2}]\!] = 3$. We will not be using this notation, instead relying upon the context to disambiguate these symbols.

If we consider a more complex language that permits the expression of functions, arrays or complex structures, then we must extend our model to include mathematical functions and structured values. For example, the denotation of:

$[\![$`function( x ) return x;`$]\!]$

is the mathematical identity function, **I**. Because we know (via the Church-Turing Thesis, [30]) that any computable function can be represented in the untyped lambda calculus, we often use lambda calculus to describe the functional values in a model domain. Applying this to the above example produces:

$[\![$`function( x ) return x;`$]\!] = \lambda x.x$

where $\lambda x.x$ is a lambda calculus expression that expresses a one-argument function that simply returns its argument. In this report, we may use the lambda calculus to express mathematical functions that are the denotations of particular programming language expressions. This mathematical lambda calculus is distinct from the programming language L_0, which will be presented in the next chapter.

There is a semantic valuation function for each syntactic domain in the language. One requirement of denotational semantics is that the valuation of any abstract syntax object is solely a function of the valuations of its syntactic components. In other words, the meaning of a piece of syntax is based only upon the constituent syntactic structure, and not upon a context outside of the syntax. The other requirement is that any given syntactic form (e.g., $plus(E, E)$) must be obtainable via a unique constructor; there cannot be two constructors that can construct the same piece of syntax. The basis for these restrictions is detailed in [22]; essentially, the first rule supports inductive reasoning, and the second rule supports unique decomposition.

Figure 1.3 summarizes the denotational semantics of L_{+-}. The model domain is simply the integers $\mathbb{Z}$ and we specify how any well-formed piece of abstract syntax can be mapped to a value in this domain. Note how the evaluation rules are recursive, where a piece of syntax's meaning is dependent upon its constituent syntactic meanings. In this case, we are specifying two distinct semantic functions, one whose domain is Num and the other whose domain is $Expr$. In both cases, however, the codomain is the set $\mathbb{Z}$.

A denotational semantics maps a source program into a value in a model domain, of which the mathematical domain of numbers and functions is one example. There are other practical and commonly used domains, however. We can consider the model of strings that are themselves source programs of another language, then our semantics is acting as a *translator*, translating each program from one language to another. If the model is assembly language or machine code programs, then our semantics acts as a compiler, translating source programs into machine code. What is common in all of these cases is that the syntax of the source is converted to a value in a way that is syntax-driven, with a meaning function for each syntactic domain.

Once we have a semantic mapping into a model domain, we can then use the model's *semantic algebra* to reason about our programs. In the case of L_{+-}, this means that we can reason about our programs in the realm of mathematics. We can determine that two distinct L_{+-} programs are *equivalent* if they have the same denotation. For example, $[\![\mathtt{1} + \mathtt{2} + \mathtt{3}]\!] = [\![\mathtt{2} + \mathtt{4}]\!]$, so we say that the two programs have the same meaning.

What I want to emphasize is that denotational semantics is typically oriented towards translation and compilation, where some or all of the computational processing occurs in the mind of the translator/compiler prior to the actual *run-time* of the program. In other words, the processing occurs during the mapping between syntax and semantics. An operational semantics as described below can be viewed as a denotational semantics translation from a source program into a model domain of machine-configurations. The semantic algebra of that model corresponds to the rewriting rules of the machine. A denotational semantics is well-suited for use in describing a pure functional language, because the model domain is simply one of values.

The research results in this report can be described primarily via operational semantics. However, denotational semantics is important because the literature often uses denotational semantics. Specifically, the original Filinski and Danvy papers on `shift` and `reset` were based upon denotational semantics. Any denotational semantics can be used to generate an operational semantics [21], although we will not be using this fact in this report.

1.1.3 Operational Semantics

Operational semantics is a powerful tool for describing, modeling, and implementing a variety of computational processes. It is based upon a discrete transition system where each step corresponds to performing some element of the entire computation. We evaluate a program operationally by first converting the program into an initial configuration, and then using the transition rules of our semantics to step from this initial configuration through a series of configurations to a final configuration, from which we extract a result. These transitions are often called rewriting rules, because two configurations differ by rewriting one or more subparts of the configuration. We now introduce an operational semantics for the L_{+-} language.

A particular phrase of L_{+-} implicitly specifies a way to compute a value by performing a series of subcomputations. For example, the concrete L_{+-} phrase $1 + (2 - 3) + 4$ corresponds to the abstract L_{+-} phrase $plus(plus(1, minus(2, 3)), 4)$, which reveals the subcomputations to be performed (the underlined expression is the active computation):

$$plus(plus(1, \underline{minus(2, 3)}), 4)$$
$$plus(\underline{plus(1, -1)}, 4)$$
$$\underline{plus(0, 4)}$$
$$5$$

The use of operational semantics allows this implicit sequence of subcomputations to be made into an explicit set of transitions between configurations. In its simplest form, we specify an operational semantics to operate directly upon the abstract syntax of our source language. The transition rules specify how an abstract syntax phrase can be evaluated into a final value in a series of rewriting steps. Later in this report, we will enhance our operational semantics so that it operates upon *configurations*, which consist of a program phrase and associated syntax that represents the state of our program.

There is great latitude and flexibility in the types of operational semantics that can be developed. For our purposes in this Introduction, we specify a very simple structured operational semantics in Figure 1.4. We are actually specifying a relation $\longmapsto$ between abstract syntax phrases such that for any syntactic object p, $p \longmapsto p'$ if p *transitions to* or *reduces to* p'. The $\longmapsto$ relation describes a single step of evaluation; we can extend it to a multi-step evaluation $\longmapsto\!\!\!\!\rightarrow$ as in Figure 1.5.

Note that the rules in Figure 1.4 constrain the $\longmapsto$ relations so that the syntax phrases that comprise the arguments to $plus$ must first be reduced to the $number(n)$ syntax (i.e., a value) before the actual transition that sums the numbers. In other words, the operational semantics imposes a left-to-right, *call-by-value* evaluation order known as *standard reduction*.

We can define an $eval(E)$ function which maps an expression in the syntactic domain $Expr$ into the particular subset of $Expr$ that has the form $value(i)$; in other words, $eval(E)$ evaluates the expression to a final value:

$$eval(E) = n \triangleq E \quad \longmapsto\!\!\!\!\rightarrow value(number(n))$$

For example, here we consider one possible reduction sequence from the concrete expression:

$$(2 * (3 + 4)) + (20 * (30 + 40))$$

to a final value 1414.

$$\begin{array}{lll}
 & (2 * (3 + 4)) + (20 * (30 + 40)) & \text{Concrete source} \\
\equiv & plus(times(2, \underline{plus(3, 4)}), times(20, plus(30, 40))) & \text{Abstract source} \\
\longmapsto\!\!\!\!\rightarrow & plus(times(2, \mathbf{7}), times(20, \underline{plus(30, 40)})) & \\
\longmapsto\!\!\!\!\rightarrow & plus(\underline{times(2, 7)}, times(20, \mathbf{70})) & \\
\longmapsto\!\!\!\!\rightarrow & plus(\mathbf{14}, \underline{times(20, 70)}) & \\
\longmapsto\!\!\!\!\rightarrow & \underline{plus(14, \mathbf{1400})} & \\
\longmapsto\!\!\!\!\rightarrow & \mathbf{1414} & \text{Final result}
\end{array}$$

1.2 Control Operators

The example language L_{+-} used in this introduction is missing several very important features that would usually be found in a practical programming language. These features can primarily be broken into the following categories:

- Functions - Define and invoke functional abstractions
- State - Read and Write storage cells
- Control - Alter the subsequent computation
- Input/Output - Read and write streams that are independent of the expression

In the remainder of this report, we will be using languages which have functions, control, and a limited form of State; we will not be discussing Input/Output features. The emphasis of this report is on *expressibility* of control features: how the control operators of one programming language can be expressed in a different programming language that lacks these particular operators but operators of equivalent expressive power.

Most programming languages have control features that allow the program to execute subexpressions conditionally, iteratively, or in various non-trivial ways that are not immediately describable in the simple reduction scheme we presented for L_{+-}. We can define synthetic programming languages to isolate and model any desired programming language feature, including these control features. This is done by adding a minimal set of *control operators* to the syntax of the lambda calculus and adding corresponding rules and structure to the semantics of the calculus.

We find that control operators within programming languages are usually organized as a small suite of operators that form a *basis* upon which more sophisticated control features can be expressed.

Exception-handling, multithreading, GOTOs and a variety of other programming language features can be described with lambda calculus extended with control operators. The well-known control operators that I have studied in my DRP include `control`, `prompt`, `shift`, `reset`, `callcc` and `abort`. For the purposes of this report, we will partition these control operators into two groups. The *abortive continuation* operators include `abort`, `callcc`, `control`, and `prompt`. The *composable continuation* operators include `shift` and `reset`. These distinctions refer to whether the captured continuation can be used as a normal function which return a value to their caller (composable), or whether invoking the continuation causes an abort to the enclosing top-level (abortive).

We will detail these control operators in their respective chapters later in this report; for this Introduction, we will only introduce the `callcc` operator to give an idea of the power of control operators, and to introduce the essential idea of *evaluation context*.

The `callcc` (**call** with **c**urrent **c**ontinuation) operator allows the current *continuation* of a subterm to be captured as a value and bound to a variable. The *continuation* of a subterm is a description of what will be evaluated after the subterm's evaluation is complete; in effect, the continuation describes the entire remaining execution state of the program at the time of capture. For example, the continuation of the subterm $(2 + 3)$ in the term $1 + (2 + 3) + 4$ can be seen as an evaluation context $1 + \square + 4$, where the $\square$ represents a *hole* into which the result of the current subcomputation is placed. After the $(2 + 3)$ is evaluated, the result 5 will be placed into the $\square$, and the evaluation of the resulting expression $1 + 5 + 4$ will continue.

When `callcc` is used, the continuation is bound to a variable that can later be used to replace the currently evaluating program with a version derived from the continuation, allowing dramatic and powerful changes to the execution of a program. For example, the program $1 + \texttt{callcc}(\lambda k.(2 + (k\ 100) + 3)) + 4$ will result in a value of 105. When the `callcc` executes, it captures its context, which is $1+\square+4$, and turns it into a value we call the continuation, which is bound to the variable k. In other words, all that `callcc` does is to turn its context into a value called a continuation, and to make that value available to its body, which is then executed normally.

It gets interesting when this continuation is invoked, as in the subphrase $(k\ 100)$ above. The `callcc`-captured continuation bound to k is an *abortive* continuation; this means that when we invoke it, rather than it executing as a normal function and returning a value, it instead aborts the current program execution, including the surrounding context, and begins executing the saved context instead. This means that when we are executing $(k\ 100)$, it does not simply return some value to be used by the surrounding context $1 + (2 + \square + 3) + 4$; instead it aborts that context and replaces it with the saved context $k = 1+\square+4$, where the $\square$ is filled in with the value 100, the parameter to k. This results in a program $1 + 100 + 4$, which is then computed normally with a result of 105.

1.3 Expressibility

We will be considering the ways that we can translate the capabilities from a *source* language into a different (but similarly powerful) set of capabilities in a *target* language. The issues that arise in this *expressibility* research reveal important and subtle behaviors of these extensions and translation. Symmetries and conservation principles arise during these translations that help us understand our programs, and observe potential pitfalls. I like to think that in the same way that scientists use experiments to reveal symmetries and conservation principles, we use tiny test programs and observe their behavior under translation or encoding.

When the source and target languages are similar (e.g., differing in only one capability), we can often *macro-express* a capability from one language in the other. This means that a subphrase in the source language can be locally translated into a subphrase in the target language. When the languages differ greatly or the capability has non-local effects, then we might need to perform a *whole-program* translation to convert a program in one language into another. This means that a subphrase in the source program might require global changes to the target program. One advantage of macro-expressibility is that it makes it easy to implement the translation as a set of ordinary functions within the target language; for example, as a library or extension to a source language.

For example, we can consider L_i, the language of fully-parenthesized *infix* arithmetic expressions (e.g., $(1 + (2 * 3)) - 4$) and the language L_p, consisting of *prefix* arithmetic expressions (e.g., $-\ (\ +\ 1\ (\ *\ 2\ 3\ )\ )\ 4$). We can formally specify a translation function $T_{i \to p}$ that can translate a program source text written in L_i into a new text written in L_p, and we can *prove* that this translation preserves the meaning and intent of the original program.

Researchers use the techniques of *translation* and *simulation* to show equivalences and relations between various languages and implementations. Translation is a specification of how a given source language text can be transformed into a target language text. Simulation is a specification of how execution of a translated text simulates the execution of a source text. A non-obvious translation requires a proof (via induction or simulation) to demonstrate its correctness.

1.4 Research Focus

The focus of this research is to see how a translation between a source set of control operators and a target set of control operators may not faithfully preserve the behavior and effects that are present in the translated expression. We study two classes of control operators that are distinguished by whether they support abortive continuations or composable continuations.

One of the primary ways that we reveal potential errors in the translation is to invent an expression in our source language and show that the behavior of the expression in the source language does not match the behavior of the translation or implementation. In the course of this research, we devised several examples that helped illuminate problems with traditional encodings.

This report will show that these errors are systematic facets of a single underlying translation error: the lack of preservation of the call-by-name semantics of the control operators. Once we understand this flaw, it appears to be a simple matter to construct an improved translation. We present improved translations for two well-known encodings.

1.5 Organization of this Report

The remainder of this report will use a variety of techniques and formalisms of programming language metatheory to describe and analyze these languages, capabilities, and the translations between them. Specifically, we will be examining how a program written in one language can be translated into a program in a different language, and how we can see whether this translation preserves the meaning of the original program.

In the next chapter, *Untyped Lambda Calculi*, we will introduce a minimal functional programming language called L_0. We will lay the groundwork necessary to see how semantic functions and metatheory can operate upon the abstract syntax. We will introduce a refinement of operational semantics known as contextual semantics, and will apply this to the semantics of L_0. This language L_0 will serve as the common subset of the other languages we will explore later in this report.

The following chapter, *Abortive Continuations*, introduces programming language features that are loosely called *abortive continuation operators*. We will describe the operators `callcc`, `abort`, `control`, and `prompt` and present the languages L_{cc} and L_{CP}. In the *Improved Encoding of Abortive Continuations*, we present the conventional encoding of L_{cc} into L_{CP}, and demonstrate via examples how this encoding is insufficient in the presence of other control effects. One of the products of our research is an improved version of this encoding, which we present in that chapter.

Next, in *Composable Continuations*, we introduce another class of control operators that are characterized by the use of continuations that can be composed and used like normal functions; i.e., they are not abortive. We will describe the `shift` and `reset` operators and the L_{SR} language.

In the chapter *Improved Encoding of Composable Continuations*, we present Filinski's encoding of L_{SR} into L_{cc}, as well as the translation of this into Standard ML. Some examples of the translation, as well as some problematic cases, are presented, motivating the improved translation that we developed during this research project. We continue by characterizing what we discovered about the Filinski implementation, illuminating where in the transformation from `shift/reset` to `callcc` the problems are introduced. Finally, we present our contribution, an improved encoding that addresses the problems.

In *Conclusion*, we summarize our results and present some of the interesting questions that we asked during the progress of this research. The motivation of this research is to understand the interexpressibility of control operators. We conclude with a discussion of some of the implications of this research, as well as related work and future directions.

Semantic Domains

n	$\in$	$\mathbb{Z}$ (Integers)	Each L_{+-} expression has a unique denotation in $\mathbb{Z}$

Semantic Valuations

$\llbracket number(n) \rrbracket$	$=$	n	The denotation of a $number()$ is the corresponding number in $\mathbb{Z}$
$\llbracket plus(E_1, E_2) \rrbracket$	$=$	$\llbracket E_1 \rrbracket + \llbracket E_2 \rrbracket$	Note how the denotation of the whole is based upon the denotations of the parts
$\llbracket minus(E_1, E_2) \rrbracket$	$=$	$\llbracket E_1 \rrbracket - \llbracket E_2 \rrbracket$	The $+$ and $-$ operators here are the mathematical addition and subtraction operations on $\mathbb{Z}$

Semantic Algebra

For L_{+-}, the semantic algebra consists simply of the addition and subtraction operations on $\mathbb{Z}$

$$+ : \mathbb{Z} \to \mathbb{Z}$$
$$- : \mathbb{Z} \to \mathbb{Z}$$

FIGURE 1.3: The Semantic valuation function $\llbracket \_ \rrbracket$ maps syntactic expressions into their denotations in the model domain. The Semantic Algebra for L_{+-} relates objects in this domain.

$$\frac{m = n_1 + n_2}{plus(number(n_1), number(n_2)) \longmapsto number(m)}\ \text{[AddNN]}$$

$$\frac{E_1 \longmapsto E_1'}{plus(E_1, E_2) \longmapsto plus(E_1', E_2)}\ \text{[AddEE]}$$

$$\frac{E \longmapsto E'}{plus(number(n_1), E) \longmapsto plus(number(n_1), E')}\ \text{[AddNE]}$$

FIGURE 1.4: The $\longmapsto$ relation in L_{+-}

$$\frac{E \longmapsto E'}{E \longmapsto\!\!\!\!\twoheadrightarrow E'}\ \text{[StepE]}$$

$$\frac{E_1 \longmapsto E_2 \qquad E_2 \longmapsto\!\!\!\!\twoheadrightarrow E_2'}{E_1 \longmapsto\!\!\!\!\twoheadrightarrow E_2'}\ \text{[StepEE]}$$

FIGURE 1.5: Inference rules for the multistep $\longmapsto\!\!\!\!\twoheadrightarrow$ relation.

$$\begin{tikzcd} L_i \arrow[r, "T_{i \to p}(-)"] & L_p \end{tikzcd}$$

FIGURE 1.6: Translation between two languages L_i and L_p

$$\begin{tikzcd}
& L_0 \arrow[dl, "\texttt{+shift+reset}"'] \arrow[dr, "\texttt{+callcc+mutable}"] & \\
L_{SR} \arrow[rr, bend left, "T_{SR\to\, CM}\text{ (Filinski)}"] \arrow[rr, bend right, "T_{SR\to\, CM}\text{ (Improved)}"'] & & L_{cc}
\end{tikzcd}$$

FIGURE 1.7: The base language L_0 (untyped λ-calculus) can be extended via the addition of the `shift` and `reset` capabilities, producing the language L_{SR}. Or, it can be extended with the `callcc` and `mutable` capabilities, producing L_{cc}.

CHAPTER II

Untyped Lambda Calculi

In the Introduction, we presented L_{+-}, which can express simple arithmetic phrases such as $3 + (4 - 2)$. The denotational semantics of L_{+-} permitted us to assign a meaning to an expression: $[\![3 + (4 - 2)]\!] = 5$. The operational semantics of L_{+-} allowed us to conclude that $3+(4-2) \Downarrow 5$ by showing that there is sequence of operational steps $3+(4-2) \longmapsto\!\!\!\!\rightarrow 5$.

We will now consider a functional language L_0 that lacks the ability to directly specify numbers and arithmetic operations. In fact, the language L_0 really has two operations: $lambda(x, e)$ and $apply(e, e)$. Figure 2.1 presents the abstract syntax for this simple untyped functional programming language. For convenience, and by convention, we will often write expressions using a concrete syntax where the symbol λ is used for the $lambda$ operation and juxtaposition is used to implicitly specify the *apply* operation. For example, the concrete $(\lambda x.A)B$ will be our concrete notation for the abstract syntax $apply(lambda(x, A), B)$. Keep in mind that for the purposes of our formalism, it is only the abstract syntax that we are speaking about; the concrete syntax is just a shorthand we use.

The $lambda$ operation allows us to abstract a piece of syntax (the body), making it conditional upon a variable (the formal argument). This operation is the same function definition capability found in most programming languages, and in all members of the λ-calculus family. Our abstract syntax characterizes a λ-expression as a value (v). For example, we can write the expression $\lambda x.\lambda y.x$, which indicates a function that takes an argument x and uses it in the construction of a new function $\lambda y.x$, which is returned. Of course, this is simply the function definition. No computation occurs until the function is applied.

We specify function application using the $apply(e_1, e_2)$ operation, which indicates that the function e_1 is to be *applied* to e_2. The syntax definition for Ł0 does not indicate what it means for an expression to be applied to another; it only tells us how to build up ever larger source programs out of $lambda$ and $apply$. We need to use a semantics to define how these source programs are interpreted or evaluated, resulting in some final value that constitutes the result of the specified computation. For our purposes in this paper, we will present an operational semantics for this language.

2.1 A Minimal Functional Language - L_0

		Syntax Domains		*Syntax Constructors*
x	$\in$	*Var* (Variables)	$::=$	$a \mid b \mid \ldots$
v	$\in$	*Val* (Values)	$::=$	$\lambda x.e$
e	$\in$	*Expr* (Expressions)	$::=$	$v \mid (e\ e)$

FIGURE 2.1: Syntax of language L_0

The operational semantics for L_0 is quite simple. The syntax constructors allow the construction of *lambda* and *apply* syntax. The only syntax destructor in our minimal language is the β-reduction rule, which removes the *lambda* syntax and exposes the body. When we use β-reduction, where each occurrence of the formal variable (x, in this case) within e_1, is replaced with e_2. The notation $e_1[e_2/x]$ is how this is indicated in Figure 2.2, where we present a first attempt at an operational semantics for L_0.

For example, consider two possible reduction sequences from the concrete expression $(\lambda x.\lambda y.(x\ y))(\lambda z.z + z)2$ to a final value 6. We will use concrete syntax throughout the reduction, although what is actually being manipulated by the rules is the equivalent abstract

Operational Semantics

$$(\lambda x.e_1)\ e_2 \quad \longmapsto \quad [e_1[e_2/x]] \qquad\qquad \beta\text{-reduction}$$

FIGURE 2.2: This operational semantics for L_0 does not enforce any order of evaluation or by-value restriction. Any available application will be reduced.

syntax. For these examples, we assume the existence of L_0 constants corresponding to $+$, 1, 2, and so on.

$$\begin{aligned}
& (\lambda x.\lambda y.(x\ y))(\lambda z.z+z)\underline{(1+2)} \\
\longmapsto\!\!\!\rightarrow\ & \underline{(\lambda x.\lambda y.(x\ y))(\lambda z.z+z)}\mathbf{3} \\
\longmapsto\!\!\!\rightarrow\ & \lambda\mathbf{y}.(\underline{(\lambda\mathbf{z}.\mathbf{z}+\mathbf{z})\mathbf{y}}))3 \\
\longmapsto\!\!\!\rightarrow\ & \underline{\lambda\mathbf{y}.(\mathbf{y}+\mathbf{y})3} \\
\longmapsto\!\!\!\rightarrow\ & \underline{\mathbf{3}+\mathbf{3}} \\
\longmapsto\!\!\!\rightarrow\ & \mathbf{6}
\end{aligned}$$

However, there is an alternate reduction sequence that is permitted by our operational semantics:

$$\begin{aligned}
& \underline{(\lambda x.\lambda y.(x\ y))(\lambda z.z+z)}(1+2) \\
\longmapsto\!\!\!\rightarrow\ & (\lambda\mathbf{y}.\underline{((\lambda\mathbf{z}.\mathbf{z}+\mathbf{z})\mathbf{y})}))(1+2) \\
\longmapsto\!\!\!\rightarrow\ & \underline{(\lambda\mathbf{y}.(\mathbf{y}+\mathbf{y}))(1+2)} \\
\longmapsto\!\!\!\rightarrow\ & \underline{(\mathbf{1}+\mathbf{2})}+(\mathbf{1}+\mathbf{2}) \\
\longmapsto\!\!\!\rightarrow\ & \mathbf{3}+\underline{(1+2)} \\
\longmapsto\!\!\!\rightarrow\ & \underline{3+\mathbf{3}} \\
\longmapsto\!\!\!\rightarrow\ & \mathbf{6}
\end{aligned}$$

The above example illustrates how our current operational semantics for L_0 permits several different reduction orders. In the above case, the order did not matter, we reached the same final answer, 6. In fact, it can be shown [22] that all reduction sequences that terminate in an irreducible value will terminate with the same irreducible value. This property is known as *confluence*, and it holds for any lambda calculus that does not have effects.

2.2 Effects and Standard Reduction

In this section, we define side-effects and show how they violate the confluence property of our simple operational semantics. We then show how we can augment our semantics to specify a well-defined order of evaluation.

An effect is some observable that is distinct from the result of a functional expression. For example, printing a string to output, changing a global variable, or raising an exception are all effects that can happen in addition to or instead of the normal return of a value from an expression. For example, consider the following ML expression:

```
val global = ref 0; (* Allocate a storage cell, initialized to 0 *)
fun foo( x, y ) = x + y + !global;
val result = foo(
                   ( global := !global + 1; 1 ),
                   ( global := !global * 2; 2 ) );
```

Without a clear specification in the ML language about the order of evaluation of arguments, we might get the answer 5 or the answer 4, depending upon whether we evaluate arguments left to right (5) or right-to-left (4). ML, and most eager languages, enforce a particular evaluation order known as *standard reduction*, which has the following properties:

- The function expression of a function application is evaluated to a value before it can be used in the application.
- The argument of a function application is evaluated to a value before it can be used in the application.
- Arguments are evaluated from left to right.
- The leftmost, outermost function application is the unique application to be reduced next.

The operational semantics we presented above in Figure 2.2 permits the β-reduction rule to be applied whenever and wherever there is an instance of function application. This non-determinism permits the various reduction sequences. In the Introduction, we specified operational semantics for L_{+-} that produced a well-defined, unique reduction order. We can use a similar refinement here to create a semantics that breaks up the β rule into several rules, which work together to ensure that our evaluation order has the standard reduction properties listed above.

We present this structural semantics in Figure 2.3. This semantics guarantees a specific order of reduction of any L_0 expression. The preconditions for the three operational rules enforce the desired evaluation order. Henceforth, we will consider L_0 to be using a standard reduction operational semantics.

$$\frac{e_1 \longmapsto e_1'}{(e_1\ e_2) \longmapsto (e_1'\ e_2)}\ [\mathit{ApplyLeft}]$$

$$\frac{e_1\ \mathit{val} \qquad e_2 \longmapsto e_2'}{(e_1\ e_2) \longmapsto (e_1\ e_2')}\ [\mathit{ApplyRight}]$$

$$\frac{e_2\ \mathit{val}}{(\lambda x.e_1\ e_2) \longmapsto (e_1[e_2/x])}\ [\mathit{Apply}]$$

FIGURE 2.3: This structural operational semantics for L_0 forces the leftmost, outermost application to be reduced, and will not reduce applications that are embedded in an enclosing lambda.

The above inference rules clarify the order of evaluation by ensuring that the $[\mathit{Apply}]$ rule can only be used when the function has been reduced to a λ-expression, and the argument has been reduced to a value. An argument can only be evaluated if the function has first been reduced to a value. However, one unfortunate aspect is that we require three operational rules to specify the single operation of function application. This is because we need to have a custom rule for each situation (i.e., computing the function, the argument, or the application). This deficiency is addressed by contextual semantics, discussed next.

2.3 Contextual Semantics

We can specify the same standard reduction order more concisely by the use of *contextual semantics* [13]. A contextual semantics adds a new syntactic component called an *evaluation context*, which acts as a focusing device that specifies where the active computation is occurring. Specifically, we extend the syntactic description of L_0 with a new category,

Ctx, which has similar syntactic structure to $Expr$, except that that any element of Ctx will have exactly one *hole*, which is usually indicated with either $[\,]$ or $\square$. The hole indicates where the active computation is occurring, the remainder of the Ctx syntax indicates the remainder of the computation.

n	$\in$	*Num* (Integers)	$::=$	$\ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$
e	$\in$	*Expr* (Expressions)	$::=$	$n \mid e + e \mid e - e$
E	$\in$	*Ctx* (Context)	$::=$	$[\,] \mid (E\ +\ e) \mid n\ E$
		$E[v_1\ +\ v_2]$	$\longmapsto$	$E[n]$ where $n = [\![v_1]\!]\ +\ [\![v_2]\!]$

FIGURE 2.4: We consolidate the syntax and semantics for L_{+-} into a single figure.

We typically consolidate the syntax and operational semantics into a single description, as we do for L_{+-} in Figure 2.4. For example, if we are computing program P_0, which is $(1 + (2 * 3)) + 4$, then we can partition this into two syntactic components:

- A context $E = 1 +\ [\,]\ + 4$, which represents a suspended computation that is waiting for a value.
- An active redex $(2 * 3)$, which is computable because 2 and 3 are values.

After computing $(2 * 3) \longmapsto 6$, we want to substitute this result into the evaluation context E to create the next configuration of our program. We write this as $E[6]$, which results in the program $P_1 \triangleq (1 + 6) + 4$. We then repeat the process, partitioning this into a context $[\,]\ + 4$ and a redex $(1 + 6)$. This yields $E[7]$ or $7 + 4$. At this point, the partition results in a context $[\,]$ and a redex $(7 + 4)$. This computation and substitution results in a final result of $[11]$, which is simply 11.

The use of evaluation contexts in our operational semantics allows us to concisely specify the order of evaluation we want our language L_0 to have. In the remainder of this report, we will usually use a single figure to describe both the syntax and the contextual operational semantics of the languages we introduce. We present the consolidated description of L_0 in Figure 2.5. We will use this format for the remainder of this report.

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Values)	v	$::=$	$\lambda x.e$
Expr (Expressions)	e	$::=$	$v \mid (e\ e)$
Ctx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E$
$[Apply]$	$E[(\lambda x.e)\ v]$	$\longmapsto$	$E[e[v/x]]$

FIGURE 2.5: We consolidate the syntax and semantics for L_0 into a single figure.

An example may help explain how to use contextual semantics. Note that each operational step requires that we first partition the program into context and computation, then we perform the computation and place the result into the hole in the context, producing the next program configuration.

Consider the program $((\lambda x.\lambda y.(x\ y))\ (\lambda z.z + z))\ 5$. We expect that the result of evaluation will be 10. Figure 2.6 illustrates the computation of this result within the L_0 operational semantics. Note that we use arithmetic here, which is not properly part of L_0, for readability purposes only; the example is intended to illustrate the use of function application, and not arithmetic.

In L_0, the use of evaluation contexts is mostly passive, where the structure of the context focuses the computation, but the context itself is never changed, it is simply constructed during the partitioning of a program. The rest of this report defines languages that extend L_0 with additional syntax to support control operators. In these extensions, the context will play a more active role, with control operators able to access the context and to change it as part of the operational step. This will prove to be a very powerful way to understand and describe control operators.

P_0	$\triangleq$	$((\lambda x.\lambda y.(x\ y))\ (\lambda z.z+z))\ 5$	Initial program
	$=$	$E_0[(\lambda x.\lambda y.(x\ y))\ (\lambda z.z+z)]$	Partition into a context and a redex where $E_0 \triangleq [\,]\ 5$
	$\longmapsto$	$E_0[\lambda y.(x\ y)[(\lambda z.z+z)/x]]$	Operation $[Apply]$
	$=$	$E_0[\lambda y.((\lambda z.z+z)y)]$	Substitution
	$=$	$(\lambda y.((\lambda z.z+z)y))\ 5$	Fill hole
	$=$	$E_1[(\lambda y.((\lambda z.z+z)y))\ 5]$ where $E_1 \triangleq [\,]$	Partition
	$\longmapsto$	$E_1[((\lambda z.z+z)y)[5/y]]$	Operation $[Apply]$
	$=$	$E_1[(\lambda z.z+z)5]$	Substitution
	$=$	$(\lambda z.z+z)5$	Fill hole
	$=$	$E_2[(\lambda z.z+z)5]$ where $E_2 \triangleq [\,]$	Partition
	$\longmapsto$	$E_2[(z+z)[5/z]]$	Operation $[Apply]$
	$=$	$E_2[5+5]$	Substitution
	$=$	$5+5$	Fill hole
	$\longmapsto\!\!\!\rightarrow$	10	Primitive operation for this example

FIGURE 2.6: Example L_0 evaluation

CHAPTER III

Abortive Continuations

In this chapter we will be looking at the suite of control operators that capture and use continuations that are *abortive*. An abortive continuation has a control effect when invoked; it aborts the default flow of control and replaces it with a new sequence of computations to be performed. We will discuss both *undelimited* and *delimited* control operators, and their interexpressibility.

As we have seen already in the previous chapters, a program can be viewed operationally as a series of steps, where each step moves the computation forward. At any point, there is an active redex (reducible expression), and an evaluation context that describes the destination of the result of the redex; i.e., what happens next. In both the L_{+-} and L_0 languages, the fate of a redex was simply to have its value computed and substituted within the hole of the evaluation context. The use of control operators permits a different outcome, where a redex containing one control operator may capture an evaluation context as a value, and a redex containing a different control operator may take such a value and make it become the current evaluation context. This enables the construction of powerful new language features such as exceptions, nondeterminism, coroutines, threading, and many more.

3.1 Undelimited Control

We will begin by describing undelimited control operators. This refers to the fact that the program is delimited only by the implicit *top-level* prompt, and that no explicit prompts are present. Later, we will present delimited continuation operators, which permit the insertion of one or more nested delimiters. This allows finer control over the context that is captured and restored via these continuations. The undelimited case can be viewed as a degenerate case of the delimited case, where there is an implicit enclosing delimiter to all programs.

3.1.1 The `callcc` operator

One of the oldest, and perhaps easiest to understand, operators is the `callcc` operator, which allows the current evaluation context to be captured and bound as a *continuation*. This continuation is a reified evaluation context, which can subsequently be used to dynamically replace the current evaluation context with the captured context. We use `callcc` by passing it a body consisting of a λ expression, which is invoked by `callcc` on our behalf. When `callcc` invokes this body expression, it will pass it the current evaluation context as a value (of syntactic type *Continuation*). This context may be ignored or used by the body of the `callcc`.

Consider the following program in L_{+-}:

$(1 + (2 * 3)) + 4$

If we consider how this program is computed operationally, we know that at some step, the program will have the evaluation context $E \triangleq 1 + \square + 4$ and the active redex $2 + 3$. In other words, the sequence of steps from that point onward will result in the value $2 + 3 => 5$ being computed and placed into the $\square$, resulting in a new program $1 + 5 + 4$. At the point where the machine is computing the atomic operation $2 + 3$, the evaluation context indicates the *rest of the computation*. We can capture this context and bind it to a variable by using the operator `callcc`:

$1 + \texttt{callcc}(fn\, k => 2 * 3) + 4$

When the `callcc` is evaluated, it will create a representation of the current evaluation context $1 + \square + 4$ and will bind this to the name k, for possible use by the body of the `callcc`. This captured evaluation context is indicated as $\langle 1 + \square + 4 \rangle$, where the angle brackets $\langle\rangle$ indicate that this is an *abortive* continuation. Subsequent evaluation and return of the $2 * 3$ computation will proceed normally; if the continuation k is not used, then the `callcc` will have no additional effect. Things get interesting when we consider using the continuation k. In ML, we use the `throw` operator to cause a captured continuation to be instantiated as the new evaluation context, and to replace the hole in this evaluation context with a value. Consider the following example in ML:

```
1 + callcc ( fn k => 2 * ( throw k 3 ) ) + 4
```

Execution proceeds as in the example above, where the continuation $1 + \square + 4$ is captured and bound to k, and the subsequent computation of 2 ∗ (\throw k 3) begins. At the point where the throw k 3 is evaluated, the evaluation context will be 1 + callcc (fn k => 2 ∗ □) + 4.

The throw k 3 will result in the saved evaluation context within k being installed as the current evaluation context 1 + \boxempty + 4, and the value 3 being used to fill the hole; in effect, this step results in the new program $1 + 3 + 4$, which is then computed to be 8. In this case, we simply used the abortive continuation as an unconditional `goto` statement which altered the flow of control. To see the more practical possibilities, consider the following example, where we use the throw k 0 to abort the computation of a product of a list of numbers (if we know one of them is 0, there is no point in examining the list further or performing any intermediate multiplications):

```
fun product numbers =
    callcc( fn k =>
                let
                    fun product_helper nil = 1
                    |   product_helper ( x :: xs ) =
                            if ( x = 0 ) then
                                throw k 0
                            else
                                x * ( product_helper xs )
                in
                    product_helper numbers
                end
    );

val z = product [1,2,3,0,5,6,7];
```

3.1.2 The L_{cc} language

For the purposes of our formal analysis, we will abstract the ML version of `callcc` slightly so that the captured continuation becomes a function-like value that can be applied to an argument without requiring the explicit `throw` keyword. In most Scheme implementations, the continuation captured by `callcc` is used by treating the continuation as a function; there is no explicit `throw` keyword. In our L_{cc} language, we will use the notation $\langle E \rangle$ to indicate an abortive continuation that is the reification of the evaluation context E. We add this `callcc` operator and the abortive continuation notation to our L_0 language to create the L_{cc} language, as defined in Figure 3.1.

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle E \rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid \mathit{callcc}(\lambda x.e)$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle E \rangle$
Ctx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E$
$E[(\lambda x.e)\ v]$		$\longmapsto$	$E[e[v/x]]$
$E[\mathit{callcc}(\lambda k.e)]$		$\longmapsto$	$E[e[\langle E \rangle/k]]]$
$E[\langle E' \rangle\ v]]$		$\longmapsto$	$E'[v]$

FIGURE 3.1: The L_{cc} language extends L_0 with the `callcc` operator and the abortive continuation operator.

Recasting the above example in our L_{cc} language shows how the continuation k is applied to its argument in the same way that a function would be (we do not use the `throw` keyword in the L_{cc} language):

$$1 + \mathit{callcc}(\lambda k.2 * (k\ 3)) + 4 => 8$$

Viewed operationally, the invocation of the $\langle E \rangle$ continuation causes the currently executing program configuration to transition to a new program consisting of $E[3]$, where $E = 1 + \Box + 4$, and therefore $E[3] = 1 + 3 + 4$. Traditional presentations of `callcc` make this abort operation explicit by providing the alternative semantics in Figure 3.2:

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid callcc(\lambda x.e) \mid \mathcal{A}_u\ e$
Ans (Answer)	e	$::=$	$\lambda x.e$
Ctx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E$

$$
\begin{array}{lll}
E[(\lambda x.e)\ v] & \longmapsto & E[e[v/x]] \\
E[callcc(\lambda k.e)] & \longmapsto & E[e[\lambda x.\mathcal{A}_u\ E[x]/k]]] \\
E[\mathcal{A}_u\ e] & \longmapsto & e
\end{array}
$$

FIGURE 3.2: This description of L_{cc} language uses an $\mathcal{A}bort$ operator instead of the abortive continuation syntax $\langle\rangle$.

We choose to not use the explicit $\mathcal{A}bort$ operation in our presentation, to avoid the confusion that may be caused by existing abort operators and to avoid the implication that the argument to abort is an ordinary expression which may be evaluated to a value. It turns out that such an implicit by-value conversion is the source of one of the encoding errors that we discuss later in this paper.

3.1.3 The $\mathcal{C}_u$ operator

One reason that `callcc` can be considered a simpler operator than the others we will be discussing is that it does nothing but passively capture the context and bind it to a name. It has no immediate control effect; it simply allows its body expression to execute. Felleisen

[7] describes an operator called $\mathcal{C}$, which was similar to `callcc` in the capture of the continuation, but had an additional control effect: the $\mathcal{C}$ operator will immediately abort the current context and execute its body as the new program. The only remnant of the aborted program will be in the form of the captured continuation. In the next section, we will be discussing the delimited version of this $\mathcal{C}$ operator; we will refer to the undelimited version here as $\mathcal{C}_u$. Similarly, we may use the name $\mathcal{K}_u$ to refer to the undelimited version of $\mathcal{K}$ that we present in the next section; however, we will usually refer to $\mathcal{K}_u$ by its more common name, `callcc`.

3.1.4 The L_{C_u} language

We define the language L_{C_u} containing the undelimited $\mathcal{C}_u$ operator in Figure 3.3.

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle E \rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid \mathcal{C}_u(\lambda x.e)$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle E \rangle$
Ctx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E$

$$\begin{aligned}
E[(\lambda x.e)\ v] &\longmapsto E[e[v/x]]\\
E[\mathcal{C}_u(\lambda k.e)] &\longmapsto E[e[\langle E \rangle/k]]]\\
E[\langle E' \rangle\ v]] &\longmapsto E'[v]
\end{aligned}$$

FIGURE 3.3: The L_{C_u} language extends L_0 with the $\mathcal{C}_u$ operator and the abortive continuation operator.

3.2 Delimited Control

The continuation that is captured by the `callcc` and $\mathcal{C}$ operators is called an *undelimited* continuation because the context that is captured includes the entire program. When an undelimited continuation is invoked, the entire program is replaced with the captured context. Effectively, there is an implicit *top-level* delimiter that surrounds the entire program (and sometimes, the system invoking the program). In this section, we consider the addition of delimiting constructs to our language, as well as control operators that capture and restore delimited context.

We can reconsider the operators $\mathcal{A}_u$, $\mathcal{K}_u$ (`callcc`) and $\mathcal{C}_u$ described previously by allowing for *delimiters* to be specified in a source expression; a delimiter can bracket a subphrase such that the scope of these operators is bounded by the delimiter, rather than going all the way to the top level of the program. Practically, delimited control enables the easy and safe expression of control effects, without the concern about affecting the entire program context. It adds modularity to control effects because an effect can be bounded such that it has no outward effect to the program outside of the delimiter.

3.2.1 The $\mathcal{C}$ (control) and $\#$ (prompt) operators

Historically, one of the first set of delimited control operators was the $\mathcal{C}$ (control) and $\#$ (prompt) operators described by Felleisen [10]. The $\#$ operator takes a subexpression as its argument and serves to delimit this subexpression from the surrounding context. This subexpression is evaluated normally, possibly resulting in other, nested $\#$ operators being executed. Thus, at some point, an execution context might have more than one delimiter in its dynamic context. As we will see below, these delimiters act to create *chambers of computation*, wherein which the abortive continuations and abortive control operators can operate, but outside of which they cannot affect.

The $\mathcal{C}$ operator can be used to abort the current execution, while capturing the delimited dynamic context to the nearest enclosing delimiter. We present this idea formally in Figure 3.4, where we introduce the language L_{CP}. The structure of our evaluation contexts has been enhanced to include two distinct types of context. The first type of context, *FCtx*, contains any prompts that may be present in the program; it acts as a delimiting context.

The other type of context, *ECtx*, acts as the delimited context; it contains no prompts, although it does contain a single hole like any evaluation context. We separate these two contexts because the $\mathcal{C}$ operator will capture only the delimited context, and the delimiting context will be uncaptured. Similarly, invoking a delimited continuation will replace the delimited context, leaving the delimiting context alone.

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle E \rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid \mathcal{C}(\lambda x.e) \mid \mathcal{K}(\lambda x.e) \mid \#e$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle E \rangle \mid$ `error:MissingReset`
FCtx (Delimiting)	F	$::=$	$[\,] \mid F\ e \mid v\ F \mid \#\ F$
ECtx (Delimited)	E	$::=$	$[\,] \mid E\ e \mid v\ E$
$E[(\lambda x.e)\ v]$		$\longmapsto$	$E[e[v/x]]$
$F[\#E[\mathcal{C}(\lambda k.e)]]$		$\longmapsto$	$F[\#(e[\langle E \rangle/k])]$
$F[\#E[\mathcal{K}(\lambda k.e)]]$		$\longmapsto$	$F[\#E[e[\langle E \rangle/k]]]$
$F[\#v]$		$\longmapsto$	$F[v]$
$F[\#E[\langle E' \rangle\ v]]$		$\longmapsto$	$F[\#[E'[v]]]$
$E[\mathcal{C}(\lambda k.e)]$		$\longmapsto$	`error:MissingReset`
$E[\mathcal{K}(\lambda k.e)]$		$\longmapsto$	`error:MissingReset`
$E[\langle E' \rangle\ v]$		$\longmapsto$	`error:MissingReset`

FIGURE 3.4: Syntax and semantics of L_{CP}

3.2.2 The Delimited $\mathcal{K}$ (`callcc`) and $\mathcal{A}$ (`abort`) operators

The $\mathcal{C}$ operator is similar to the `callcc` operator in that it captures an abortive continuation. However, the $\mathcal{C}$ operator is itself abortive; it has a control effect which is to immediately abandon the current delimited context, replacing it with the body of the $\mathcal{C}$. This is distinct from `callcc`, which captures the continuation but does not alter the current context. We use an operator $\mathcal{K}$ to specify a `callcc`-like operator that does not abort the current context, but only captures the delimited context; ordinarily, `callcc` captures the entire program context, ignoring delimiters. We define a language L_{AK} in Figure 3.5 which contains the $\mathcal{K}$ and $\mathcal{A}$ operators. This is similar to the language L_{cc} defined in Figure 3.1, with the addition of delimiters and a modification to the operational rules and evaluation contexts to support delimited contexts. As is conventional in the delimited case, we use the notation $\mathcal{K}$ instead of `callcc`.

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle E \rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid \mathcal{K}(\lambda x.e) \mid \mathcal{A}\ e \mid \#e$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle E \rangle \mid$ `error:MissingReset`
FCtx (Delimiting)	F	$::=$	$[\,] \mid F\ e \mid v\ F \mid \#\ F$
ECtx (Delimited)	E	$::=$	$[\,] \mid E\ e \mid v\ E$

$$
\begin{array}{lcl}
E[(\lambda x.e)\ v] & \longmapsto & E[e[v/x]] \\
F[\#E[\mathcal{K}(\lambda k.e)]] & \longmapsto & F[\#E[e[\langle E \rangle/k]]] \\
F[\#v] & \longmapsto & F[v] \\
F[\#E[\langle E' \rangle\ v]] & \longmapsto & F[\#[E'[v]]] \\
E[\mathcal{K}(\lambda k.e)] & \longmapsto & \texttt{error:MissingReset} \\
E[\langle E' \rangle\ v] & \longmapsto & \texttt{error:MissingReset}
\end{array}
$$

FIGURE 3.5: Syntax and semantics of L_{AK}

An example maybe be helpful to distinguish the two operators $\mathcal{C}$ and $\mathcal{K}$. Consider the following programs:

$$1 + \#(2 + \mathcal{K}(\lambda k.k\ 3)) + 4$$
$$1 + \#(2 + \mathcal{C}(\lambda k.k\ 3)) + 4$$

3.2.3 Expressing $\mathcal{C}_u$ as $\mathcal{K}_u$ and $\mathcal{A}_u$

It is possible to express the $\mathcal{C}_u$ operator using $\mathcal{K}_u$ (`callcc`) and an undelimited abort operator, $\mathcal{A}_u$ [7]:

$$\mathcal{C}_u(\lambda k.e) \triangleq \mathcal{K}_u(\lambda k.\mathcal{A}_u\ e)$$

In this particular encoding, we intend for the $\mathcal{A}_u$ operator to be non-strict. A strict version of $\mathcal{A}_u$ would result in a space leak.

Verify the [7] reference

3.3 Summary

We presented abortive continuations, in both their delimited and undelimited forms. In a later chapter, *Improved encodings of Abortive Continuations*, we discuss the converse encoding of `callcc` in terms of $\mathcal{C}_u$, where strictness again introduces a space leak and other problems. One contribution of our research is to improve this encoding, which we present in that chapter.

CHAPTER IV

Composable Continuations

In the previous chapter, we discussed abortive continuations. These were characterized by the fact that invoking the continuation abandoned the current evaluation context. We divided these into delimited and undelimited forms. In this chapter, we consider a different type of delimited control operator, one which provides for *composable* continuations. This means that we can invoke the continuation like a normal function that simply returns a value, rather than the continuation aborting the current computation. The body of this continuation function contains the captured context; when the continuation is invoked with a value, the hole in the context will be filled with the argument value and the resulting expression will be computed and returned as the result.

4.1 The `shift` and `reset` operators

Danvy and Filinski [5] introduced the `shift` and `reset` operators to implement a monadic programming pattern within any language supporting these operators. Whatever the original motivation, these operators have proven to be a useful basis for constructing more complex control structures. In much the same way as the `prompt`/`control` pair acts to delimit and capture context, the `reset`/`shift` pair of operators acts to delimit and capture context. The only real difference is that the context that is captured by `shift` is made available as an ordinary composable function. For this report, we will often use the generic term *prompt* to refer to both the `prompt` and `reset` operators, which we indicate with $\#$ in both L_{CP} and L_{SR}.

For example, consider the following expressions, one which uses $\mathcal{S}$ and the other $\mathcal{C}$:

$2 + \#(1 + \mathcal{C}(\lambda k.(k(k\ 2)))) \longmapsto\!\!\!\!\rightarrow 5$

$2 + \#(1 + \mathcal{S}(\lambda k.(k(k\ 2)))) \longmapsto\!\!\!\!\rightarrow 6$

In both cases, the captured delimited context is $1+\square$. In the case of the abortive operator $\mathcal{C}$, the inner invocation of $(k\ 2)$ never returns, instead replacing the current delimited context with the value 2, causing the next program configuration to be $2 + (1 + 2) = 5$. The outermost invocation of k never occurs. However, in the case of $\mathcal{S}$, the inner invocation of $(k\ 2)$ is tantamount to invoking a normal function $\lambda x.1 + x$, which returns $1 + 2 = 3$. This ordinary result value is then passed to the outer k: $(k\ 3)$. This then returns $1+3 = 4$, which is used to replace the delimited context $2 + \square$, resulting in $2 + 4 = 6$, the final result.

Our notation distinguishes these cases by using the syntax $\langle E\rangle$ to indicate an abortive continuation (as captured by $\mathcal{C}$ and $\mathcal{K}$, and $\langle\!\langle E\rangle\!\rangle$ to indicate a composable continuation (as captured by $\mathcal{S}$). Figure 4.1 presents the abstract syntax for a functional programming language containing the `shift` ($\mathcal{S}$) and `reset` ($\#$) operators.

4.2 Composable continuations can be used without a reset

Because composable continuations look and act like regular functions, there should be no restriction on their context of use. Earlier in our research, we had thought that using a `shift`-captured continuation would require that we have an enclosing prompt. This was encoded as two rules: one that safely handled the composable continuation if a reset was present, and another rule that produced MissingReset if the use of a composable continuation had no reset.

$$\begin{array}{rcl} F[\langle\!\langle E\rangle\!\rangle\ v] & \longmapsto & F[\#E[v]] \\ E[\langle\!\langle E'\rangle\!\rangle\ v] & \longmapsto & \texttt{error:MissingReset} \end{array}$$

However, we have since corrected the semantics to allow such use. Composable continuations are indistinguishable from ordinary functions. For example, the following L_{SR} expression returns a composable continuation from within a `shift` and then applies this continuation to an argument that lies outside of any enclosing prompt.

$(\#\texttt{shift}(\lambda k.k))\ 0$

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle\!\langle E \rangle\!\rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid S(\lambda x.e) \mid \#e$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle\!\langle E \rangle\!\rangle \mid$ `error:MissingReset`
FCtx (Context)	F	$::=$	$[\,] \mid F\ e \mid v\ F \mid \#\ F$
ECtx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E$

$$\begin{array}{lcl}
E[(\lambda x.e)\ v] & \longmapsto & E[e[v/x]] \\
F[\#E[\mathcal{S}(\lambda k.e)]] & \longmapsto & F[\#(e[\langle\!\langle E \rangle\!\rangle/k])] \\
F[\#v] & \longmapsto & F[v] \\
E[\langle\!\langle E' \rangle\!\rangle\ v] & \longmapsto & E[\#E'[v]] \\
E[\mathcal{S}(\lambda k.e)] & \longmapsto & \texttt{error:MissingReset}
\end{array}$$

FIGURE 4.1: Syntax and semantics of the `shift/reset` language L_{SR}

Using our original (incorrect) operational semantics, we would expect an uncaught `MissingReset` exception via the following reduction sequence:

$$\begin{array}{lll}
 & (\#\texttt{shift}(\lambda k.k))\ 0 & \text{Original expression} \\
\longmapsto & (\#(k[\langle\!\langle [\,] \rangle\!\rangle/k]))\ 0 & \\
= & (\#\langle\!\langle [\,] \rangle\!\rangle)\ 0 & \\
\longmapsto & \langle\!\langle [\,] \rangle\!\rangle 0 & \texttt{MissingReset}
\end{array}$$

After removing the erroneous rule from our semantics, we see the following reduction:

$$\begin{array}{ll}
 & (\#\texttt{shift}(\lambda k.k))\ 0 \quad \text{Original expression} \\
\longmapsto & (\#(k[\langle\!\langle[\,]\rangle\!\rangle/k]))\ 0 \\
= & (\#\langle\!\langle[\,]\rangle\!\rangle)\ 0 \\
\longmapsto & \langle\!\langle[\,]\rangle\!\rangle\ 0 \\
\longmapsto & \#0 \\
\longmapsto & 0
\end{array}$$

which produces a value of 0.

In the process of convincing ourselves that composable continuations are indistinguishable from ordinary functions, we evaluated the above example using several different techniques, which we present here and in the Appendix.

4.2.1 Confirmation via ML

We wrote an ML version of the example using both Filinski's implementation of shift/reset, as well as our improved implementation. Both implementations behaved as predicted; using a captured composable continuation is completely safe. See the Appendix *ML Implementations and Examples* for the implementations used. The example *Composable Continuation Usable Outside of Reset* has the test example we used. Both the original Filinksi implementation and our improved implementation support this behavior.

4.2.2 Confirmation via the metacontinuation ECPS semantics

In the *Representing Monads* [11] and its predecessor *Abstracting Control* [5], there is a denotational semantics presented called Extended Continuation Passing Style (ECPS). This denotational semantics is the original specification of the `shift` and `reset` operators; the operational semantics that we use was developed later. Out of curiousity and a desire for completeness, I encoded the example $(\#\texttt{shift}(\lambda k.k))\ 0$ into the ECPS and performed the reduction, confirming once again that composable continuations act like ordinary functions.

The detailed translation and reduction are presented in Appendix *Extended Continuation Passing Style*.

4.2.3 Confirmation via the Danvy Abstract Machine

As yet another way to confirm this result, and an opportunity to explore a different specification mechanism, I used an abstract machine operational semantics to encode the example program $(\#\texttt{shift}(\lambda k.k))0)$. The abstract machine for `shift/reset` was originally defined by Biernacka, Beirnacki, and Danvy [1]. We are actually using the revised version presented in [3, 2]. Using this machine produced the expected result. The detailed evaluation sequence and the definition of the machine is in the Appendix, *The shift/reset abstract machine*.

4.3 Summary

We have shown how composable continuations differ from their abortive counterparts, and how the `shift` and `reset` operators can be used to perform similar control operations to their abortive counterparts `control` and `prompt`. In the next two chapters, we will be illustrating how these two suites of operators can be encoded into languages that lack these operators, and some of the issues that arise.

CHAPTER V

Improved Encoding of Abortive Continuations

As we showed in *Abortive Continuations*, we can encode the undelimited $\mathcal{C}_u$ in a language with `callcc` and `abort`. In this chapter, we discuss the reverse, where we encode `callcc` in a language containing the $\mathcal{C}_u$ operator. In the first section, we discuss a traditional encoding and describe some of its problems. In the second section, we present an improved encoding, one of the contributions of this research.

5.1 Traditional Encoding of `callcc` as $\mathcal{C}_u$

We begin by considering a well-known encoding of `callcc` into $\mathcal{C}_u$, shown below:

$$\texttt{callcc}(\lambda k.e) \triangleq \mathcal{C}_u(\lambda k.k\ e)$$

Notice how the context that would normally remain behind with `callcc` is first removed by the $\mathcal{C}$ operator, and then reestablished by the application of the captured continuation in $k\ e$. In the example below, we compare the evaluation of a program and its encoding. The original program:

$$\begin{array}{ll} & (1 + \texttt{callcc}(\lambda k.(k\ 2))) + 4 \\ \longmapsto & (1 + (k\ 2)) + 4 \text{ where } k = \langle (1 + \Box) + 4 \rangle \\ \longmapsto & ((1 + \Box) + 4)[2/\Box] \\ = & ((1 + 2) + 4) \\ \longmapsto & 3 + 4 \\ \longmapsto & 7 \end{array}$$

The encoded version of the above example:

$$\begin{array}{ll} & (1 + \mathcal{C}(\lambda k.(k(k\ 2))) + 4 \\ \longmapsto & k(k\ 2) \text{ where } k = \langle (1 + \Box) + 4 \rangle \\ \longmapsto & ((1 + \Box) + 4)[2/\Box] \\ = & ((1 + 2) + 4) \\ \longmapsto & 3 + 4 \\ \longmapsto & 7 \end{array}$$

In the above example, both the original and the encoded version of the program result in the same value, although the paths by which they reach that result differ. We can choose to observe the amount of space overhead used during a computation, which is a practical consideration in a real computer. When we observe the space our programs use, then we see that our encoding has a problem: we can construct programs where the encoded version has a dramatically different space overhead than the original. This can be so significant as to make the encoding unusable for large problems.

We present an example of such a *space leak* below. Consider the following definition of a $loop$ function:

$$loop\ 1 = 1$$
$$loop\ n = \texttt{callcc}(\lambda k.loop(n-1))$$

Using this definition, the program $loop\ 3$ has the following reduction sequence:

$$\begin{array}{ll} & loop\ 3 \\ \longmapsto & \texttt{callcc}(\lambda k.loop(3-1)) \\ \longmapsto & loop(3-1) \\ \longmapsto & \texttt{callcc}(\lambda k.loop(2-1)) \\ \longmapsto & loop(2-1) \\ \longmapsto & 1 \end{array}$$

Using the encoding provided above, the $loop$ definition becomes:

$$loop\ 1 = 1$$
$$loop\ n = \mathcal{C}(\lambda k.k\ (loop(n-1)))$$

and the reduction sequence for $loop\ 3$ is:

$$\begin{array}{lll}
 & \mathit{loop}\; 3 & \\
\longmapsto & \mathcal{C}_u(\lambda k.k\; \mathit{loop}(3-1)) & \\
\longmapsto & k\; \mathit{loop}(3-1) & \text{where } k = \langle \Box \rangle \\
\longmapsto & k\; \mathit{loop}(2) & \text{Notice how the argument to } k \\
 & & \text{is being reduced} \\
\longmapsto & k\; \mathcal{C}_u(\lambda k.k\; \mathit{loop}(2-1)) & \\
\longmapsto & k\; (k\; \mathit{loop}(2-1))) & \\
\longmapsto & k\; (k\; \mathit{loop}(1))) & \\
\longmapsto & k\; (k\; 1)) & k \text{ is an abortive continuation} \\
\longmapsto & 1 &
\end{array}$$

The result is the same, but the encoded result suffers from a space overhead that is in this case proportional to the value of the initial argument to $loop$. Notice how the reduction sequence builds up a sequence of $k\ (k\ \ldots)$ until it finally evaluates the base case of $loop\ 1$, where it finally begins unwinding the stack of k-invocations. This means that not only does the encoded version suffer from a space problem, but that the number of continuation invocations is different. In other words, if we choose to make space or number of invocations an *observable*, we can distinguish the original program from its encoding.

As a practical example, when I run the program $loop\ 1000000$ using the traditional encoding of $\mathcal{C}$ into ML, I use an enormous amount of memory (on the order of gigabytes), which drastically affects the elapsed time performance of what should be a very fast loop, turning it into a several-minute rather than a several-millisecond operation.

This encoding also suffers in the presence of other effects, such as exceptions. Consider the following program, and its reduction, which produces 0 as its answer:

$$\begin{array}{ll}
 & \texttt{callcc}(\lambda k => \texttt{raise Fail})\ \texttt{handle Fail} => \texttt{0} \\
\longmapsto & (\texttt{raise Fail})\ \texttt{handle Fail} => \texttt{0} \\
\longmapsto & 0
\end{array}$$

When we contrast the above with the encoded version below, we see a difference; the encoded version produces an uncaught exception `Fail`.

$$\begin{array}{ll}
 & \mathcal{C}_u(\lambda k => k\ (\texttt{raise Fail}))\ \texttt{handle Fail} => \texttt{0} \\
\longmapsto & k\ (\texttt{raise Fail}) \text{ where } k = \langle \Box\ \texttt{handle Fail} => \texttt{0} \rangle \\
\longmapsto & \text{Uncaught Exception: } Fail
\end{array}$$

This error is due to the fact that the invocation of k in the second example is strict, forcing the argument `raise Fail` to be evaluated *before* the control transfer via the k-invocation occurs. The whole point of the k-invocation in the encoding is to reestablish the context that was deleted by $\mathcal{C}_u$, so that we can emulate `callcc`, which does not delete the context. However, the strictness of the k forces the evaluation to occur prior to us reestablishing this context. In this example, the consequence is that the exception handler is not in place when the exception is raised. Both of the types of problems above can be attributed to the encoding improperly evaluating the body of the control operator prior to establishing the correct execution context. In the next section, we introduce our contribution, an improved encoding that correctly establishes the context before executing the encoded body.

5.2 An Improved Encoding

In the encoding above, the original body e_{body} of the `callcc`$(\lambda k.e_{body})$ is encoded as an expression which is being passed as an argument to the abortive continuation k in the encoding. Because the continuation k is defined as a strict operation, this body expression is evaluated before the continuation k is able to abort the current context and establish the continuation's saved context. This results in the body e being evaluated in the unintended pre-abort context, which can result in the problems above.

Once we were able to characterize these problems as being due to strictness in the encoded version of our operators, it was straightforward to conceive and implement a solution. The solution is to encode the body e_{body} in such a way that we can defer its execution until the proper context has been established by the encoded operator. This is done by encoding the body e_{body} in `callcc`$(\lambda k.e_{body})$ as a function $\lambda \_.e_{body}$. The identifier $\_$ in this case means that we will not use the argument to this function in e_{body}; it is simply a placeholder or dummy variable. The functional abstraction is being used to delay the execution of e_{body}, and not to perform any parameter substitution. When we use a function in this way to delay execution, we call it a *thunk* and the process of delaying code is called *thunking*. We will usually use a dummy argument indicated as $()$ (often called *unit* or *nil*) to invoke a thunk function. This programming pattern is often known as a *delay/force* transformation, where the construction of a thunk delays the code, which is later forced by the application of the thunk to $()$.

Here is our improved encoding, where the body has been replaced with a thunk, and context now includes the execution of a thunk:

$$\texttt{callcc}(\lambda k.e) \triangleq \mathcal{C}(\lambda k'.(k'\ \lambda\_.e[\lambda x.(k'\ \lambda\_.x)/k]))\ ()$$

Or alternatively, using a reduction instead of an explicit substitution:

$$\texttt{callcc}(\lambda k.e) \triangleq \mathcal{C}(\lambda k'.(k'\ \lambda\_.((\lambda k.e)(\lambda x.(k'\ x)))))\ ()$$

Notice that the context that is captured by the encoded $\mathcal{C}$ operator will be $\langle [\,]\ ()\rangle$, which means that the context is expecting a function that takes a dummy argument (indicated as $()$ here). This gives us the ability to pass code to the context, filling the hole and causing the execution of the code to happen in the context, rather than happening before the context is established.

Given our improved encoding, we can reconsider the space leak example above. The $loop$ definition under the new encoding becomes:

$$\begin{aligned} loop\ 1 &= 1 \\ loop\ n &= \mathcal{C}(\lambda k'.k'\ \lambda\_.loop(n-1)[(k\ \lambda\_.x)/k]))\ () \\ &= \mathcal{C}(\lambda k'.k'\ \lambda\_.loop(n-1))\ () \end{aligned}$$

and the reduction sequence for $loop\ 3$ is:

$$\begin{array}{lll} & loop\ 3 & \\ \longmapsto & \mathcal{C}(\lambda k'.k'\ \lambda\_.loop(3-1))\ () & \\ \longmapsto & k'\ \lambda\_.loop(3-1) & \text{where } k' = \langle \Box\ ()\rangle \\ \longmapsto & \lambda\_.loop(3-1)\ () & \\ \longmapsto & loop(3-1) & \\ \longmapsto & loop(2) & \\ \longmapsto & \mathcal{C}(\lambda k'.k'\ \lambda\_.loop(2-1))\ () & \\ \longmapsto & k'\ \lambda\_.loop(2-1) & \text{where } k' = \langle \Box\ ()\rangle \\ \longmapsto & \lambda\_.loop(2-1)\ () & \\ \longmapsto & loop(2-1) & \\ \longmapsto & loop(1) & \\ \longmapsto & 1 & \end{array}$$

As can be seen above, the improved encoding has a fixed space overhead, independent of the $loop$ parameter. We do not accumulate a stack of k-invocations.

If we revisit the exception example above, we can encode the program:

$$\texttt{callcc}(\lambda k \texttt{ => raise Fail}) \texttt{ handle Fail => 0}$$

using our improved encoding as:

$$\begin{array}{ll}
 & (\mathcal{C}_u(\lambda k' \texttt{ => } k'\ \lambda\_.(\texttt{raise Fail})[\lambda \texttt{x}.(\texttt{k}'\ \lambda\_.\texttt{x})/\texttt{k}])\ ())\ \texttt{handle Fail => 0} \\
= & (\mathcal{C}_u(\lambda k' \texttt{ => } k'\ \lambda\_.(\texttt{raise Fail}))\ ())\ \texttt{handle Fail => 0} \\
\longmapsto & (k'\ \lambda\_.(\texttt{raise Fail})) \\
 & \text{where } k' = \langle(\square\ ())\ \texttt{handle Fail => 0}\rangle \\
\longmapsto & (\lambda\_.(\texttt{raise Fail}))\ ())\ \texttt{handle Fail => 0} \\
\longmapsto & (\texttt{raise Fail})\ \texttt{handle Fail => 0} \\
\longmapsto & 0
\end{array}$$

As shown above, our improved encoding results in 0, which is faithful to the original semantics `callcc`, rather than the erroneous uncaught exception `Fail` that we obtained using the traditional encoding.

5.3 A Proof of Correctness

The final step in establishing our improved encoding is to prove a theorem showing that for any program $P \in L_{cc}$, it will produce the same result as its encoding $[\![P]\!] \in L_{C_u}$. We also wish to know that an encoding diverges (does not halt) if its source expression diverges, and that if an encoding diverges, then its source diverges. We use the notation $e \Uparrow$ to indicate that there is no value v such that $e \longmapsto\!\!\!\!\rightarrow v$.

Because an encoding may take a different number of steps to reach an answer, and because the expression in the encoding is not identical with the source expression, we define a *simulation* relation between source and encoding. This relation is indicated as $\lceil e \rceil \sim e$, which means that the encoding of e, called $\lceil e \rceil$, simulates e.

This simulation relation is defined inductively on the structure of the source language. We can prove a theorem which establishes that for each configuration e we have in a source reduction sequence, there is a configuration (expression) in the encoded expression's reduction sequence that simulates e, and vice versa.

Theorem 5.3.1. *Given source expression e_0, the following properties hold:*

1. $\forall e_m . e_0 \longmapsto\!\!\!\!\rightarrow e_m \Rightarrow \exists e'_n . \lceil e_0 \rceil \longmapsto\!\!\!\!\rightarrow e'_n \wedge e'_n \sim e_m$

2. $\forall e_n . \lceil e_0 \rceil \longmapsto\!\!\!\!\rightarrow e'_n \Rightarrow \exists e_m . e_0 \longmapsto\!\!\!\!\rightarrow e_m \wedge e'_n \sim e_m$

3. $e_0 \Uparrow \iff \lceil e_0 \rceil \Uparrow$

5.4 Summary

We described the two stages of the traditional Filinski encoding, and showed problems with both stages. The $L_{SR} \rightarrow L_{AK}$ encoding had space and effect problems which we addressed using a delay/force transformation. The problems with the second stage were addressed because the new encoding prevented the corruption of the metacontinuation stack.

The appendix to this report contains Standard ML implementations of both the original and the improved versions of these operators, as well as implementations of the above test cases.

CHAPTER VI

Improved Encoding of Composable Continuations

In previous chapters, we have introduced abortive and composable continuations and their associated operators `callcc`, `control`, `prompt`, `abort`, `shift` and `reset`. These operators capture and replace context with respect to either an explicit delimiter called a prompt (#) or an implicit delimiter called the top level. The previous chapter dealt with the encoding of abortive continuation operators, the particular problems that occur, and our research contribution in the form of a new encoding. In this chapter, we study an analogous problem in the realm of composable continuation operators.

We consider the encoding of the delimited operators `shift` and `reset` within a language containing only the undelimited operator `callcc` and a single mutable storage location. Filinski's *Representing Monads* [11] paper revealed one possible version of this encoding. The first section of this chapter will describe Filinski's original encoding as a two-stage encoding; the first stage from L_{SR} to L_{AK} and the second stage from L_{AK} to L_{cc}. The second section will illustrate some problems with the encoding and these will motivate our improved encoding, which we present in the third section of this chapter.

6.1 A traditional encoding of `shift` and `reset`

The operators `shift` and `reset` can be encoded into a language that contains only an undelimited `callcc`, and a single mutable storage variable. This was shown by Andrzej Filinski in his important *Representing Monads* paper, where he develops this encoding in two stages.

The first stage of the encoding is to transform a program containing the composable control operators `shift` and `reset` into an equivalent program that uses the abortive delimited operators $\mathcal{K}$, $\mathcal{A}$, and $\#$. In the second stage, this program is translated into an equivalent program that uses only undelimited `callcc` and a single mutable storage cell. Formally, the first stage will be translating a program P from the language L_{SR} into the language L_{AK}, while the second stage will translate from L_{AK} into L_{cc}. This is quite a dramatic result, and has provided the practical ability to implement `shift` and `reset` within those languages containing only `callcc`, including Standard ML, Scheme, Haskell, and Ruby.

In our source language L_{SR}, an invocation of a composable continuation $(k\ x)$ can be encoded as an abortive continuation in L_{AK} by surrounding the invocation with a prompt, $\#(k\ x)$. This ensures that when the abortive continuation in L_{AK} is invoked, it will only abort the newly delimited context, passing its result back as if it were a normal composable function.

In the following example, we wrap one invocation of the abortive continuation k with a prompt; this ensures that the invocation returns a value, which we pass to the second invocation, which then aborts to the outermost prompt:

$$\begin{array}{ll}
 & 2 + \#(1 + \mathcal{C}(\lambda k.k\ (\#(k\ 2)))) \\
\longmapsto & 2 + \#(\langle 1 + [\,]\rangle\ (\#(\langle 1 + [\,]\rangle\ 2))) \\
\longmapsto & 2 + \#(\langle 1 + [\,]\rangle\ (1 + 2)) \\
\longmapsto & 2 + \#(\langle 1 + [\,]\rangle\ 3) \\
\longmapsto & 2 + (1 + 3) \\
\longmapsto & 2 + 4
\end{array}$$

Building upon this ability to encode a composable invocation, we can provide the traditional encoding of `shift` in terms of $\mathcal{C}$, which transforms abortive invocations into composable ones:

$$\mathcal{S}(\lambda k.e) \triangleq \mathcal{C}(\lambda k.e[\lambda x.\#(k\ x)/k]))$$

There is an equivalent encoding in L_{AK}, which we will be using in our subsequent descriptions. In this L_{AK} encoding, we passively capture the continuation with $\mathcal{K}$ and then immediately abort to remove the surrounding context, as demanded by $\mathcal{S}$:

$$\mathcal{S}(\lambda k.e) \triangleq \mathcal{K}(\lambda k.\mathcal{A}\; e[\lambda x.\#(k\; x)/k]))$$

In the above encoding, the $\mathcal{A}$ is the delimited abort operator described in the *Abortive Continuations* chapter. Note that this $\mathcal{A}$ is not strict, it does not evaluate its argument expression to a value; it instead uses the argument expression to replace the delimited context, where it will be evaluated. However, in the process of realizing the encoding in a call-by-value language such as ML, Filinski treats this operator as a strict abort, which we will call $\mathcal{A}_s$. The semantics of this strict abort require that its argument is evaluated to a value prior to performing the abort. Formally:

$$F[\#E[\mathcal{A}_s\; v]] \longmapsto F[\#v]$$

making the encoding:

$$\mathcal{S}(\lambda k.e) \triangleq \mathcal{K}(\lambda k'.\mathcal{A}_s\; e[\lambda x.\#(k'\; x)/k]))$$

The consequences of this choice are that effect reordering can occur because of the strictness of the abort operator. We will expand upon this in the next section, after we present the second stage of the encoding below.

We showed above how L_{SR} programs are encoded into L_{AK} via the traditional encoding; we now examine the second stage, which is the encoding of a L_{AK} program into the language L_{cc} containing only the `callcc` operator and a single storage variable. This is done by using a single global variable mk to store an undelimited continuation corresponding to the most recent prompt. The delimited $\mathcal{K}$ will be encoded as `callcc`, the prompt ($\#$) operator will update mk, and the $\mathcal{A}_s$ operator will read mk.

We extend the language L_{cc} presented in the *Abortive Continuations* chapter with the capability to read and write a storage variable mk. We present the syntax and semantics of this augmented version of L_{cc} in Figure 6.1. Notice that a new syntactic form has been introduced called *Configuration*, which is a pairing of an *Expression* which is the current program, and a *Value* which is the current contents of the mk storage variable. The operational semantics has been extended to specify how a *Configuration* transitions to its successor.

The second stage of the encoding, that from L_{AK} to L_{cc}, follows:

$$\begin{array}{lll}
\mathcal{K} & \triangleq & \texttt{callcc} \\
\mathcal{A}_s & \triangleq & !mk \\
\#e & \triangleq & \texttt{callcc}(\lambda k.\texttt{let}\, m = !mk \,\texttt{in} \\
 & & \qquad (mk := \lambda x.(mk := m; k\ x); \\
 & & \qquad \mathcal{A}_s\ e))
\end{array}$$

We present the combined two-stage encoding as:

$$\begin{array}{lll}
\mathcal{A}_s & \triangleq & !mk \\
\#e & \triangleq & \texttt{callcc}(\lambda k.\texttt{let}\, m = !mk \,\texttt{in} \\
 & & \qquad (mk := \lambda x.(mk := m; k\ x); \\
 & & \qquad \mathcal{A}_s\ e)) \\
\mathcal{S}(\lambda k.e) & \triangleq & \texttt{callcc}(\lambda k.(\mathcal{A}_s\ e[\lambda x.\#(k\ x)/k]))
\end{array}$$

Given these two encodings that are based upon Filinski's encodings, we know that any program in L_{SR} can be translated into an equivalent program in L_{cc}. Practically, this enables the implementation of `shift` and `reset` in conventional functional programming languages such as Scheme and ML, both of which contain mutable storage and `callcc` features. In the next section, we show some problems with each of the two stages of this encoding.

6.2 Problems with the Traditional Encoding

Both stages of the encoding above have some problems, which we will describe here. Using the $L_{SR} \rightarrow L_{AK}$ encoding (first stage), we can observe a space leak similar to that in the previous chapter. Consider the following definition of a *loop* function:

$$\begin{array}{l}
\mathit{loop}\ 1 = 1 \\
\mathit{loop}\ n = \mathcal{S}(\lambda k.\mathit{loop}(n-1))
\end{array}$$

Using the operational semantics of L_{SR}, we can consider the reduction of the program $\mathit{loop}\ 3$. Filinski points out that in order to compute an expression, it should be wrapped in an outermost reset; so our actual initial program is $\#(\mathit{loop}\ 3)$

Var (Variables)	x	$::=$	$a \mid b \mid \ldots$
Val (Value)	v	$::=$	$x \mid \lambda x.e \mid \langle E \rangle$
Expr (Expression)	e	$::=$	$v \mid (e\ e) \mid \mathit{callcc}(\lambda x.e) \mid !mk \mid mk := e \mid \texttt{MissingReset}$
Ans (Answer)	e	$::=$	$\lambda x.e \mid \langle E \rangle$
Ctx (Context)	E	$::=$	$[\,] \mid E\ e \mid v\ E \mid mk := E$
Cfg (Configuration)	c	$::=$	$e, v \mid \texttt{MissingReset}, v$

$$\begin{array}{lll}
E[(\lambda x.e)\ v], v' & \longmapsto & E[e[v/x]], v' \\
E[\mathit{callcc}(\lambda k.e)], v' & \longmapsto & E[e[\langle E \rangle/k]], v' \\
E[\langle E' \rangle\ v]], v' & \longmapsto & E'[v], v' \\
E[!mk], v' & \longmapsto & E'[v'], v' \\
E[mk := v], v' & \longmapsto & E'[v], v \\
E[\texttt{MissingReset}], v' & \longmapsto & \texttt{MissingReset}, v'
\end{array}$$

FIGURE 6.1: The L_{cc} language features `callcc` and a single mutable storage variable mk

$$\begin{array}{ll}
 & \#(\mathit{loop}\ 3) \\
\longmapsto & \#(\mathcal{S}(\lambda k.\mathit{loop}(3-1))) \\
\longmapsto & \#(\mathit{loop}(3-1)) \\
\longmapsto & \#(\mathit{loop}(2)) \\
\longmapsto & \#(\mathcal{S}(\lambda k.\mathit{loop}(2-1))) \\
\longmapsto & \#(\mathit{loop}(2-1)) \\
\longmapsto & \#(\mathit{loop}(1)) \\
\longmapsto & \#(1) \\
\longmapsto & 1
\end{array}$$

When we consider the encoded version in L_{AK}, we notice the same space leak that we observed when encoding abortive continuations in the previous chapter. The *loop* definition under this L_{AK} encoding becomes:

$$\begin{array}{lcl} \mathit{loop}\, 1 & = & 1 \\ \mathit{loop}\, n & = & \mathcal{K}(\lambda k'.\mathcal{A}_s\, \mathit{loop}(n-1)[(k'\, x)/k])) \\ & = & \mathcal{K}(\lambda k'.\mathcal{A}_s\, \mathit{loop}(n-1)) \end{array}$$

and the reduction sequence for *loop* 3 is:

$$\begin{array}{ll} & \#(\mathit{loop}\, 3) \\ \longmapsto & \#(\mathcal{K}(\lambda k'.\mathcal{A}_s\, \mathit{loop}(3-1))) \\ \longmapsto & \#(\mathcal{A}_s\, \mathit{loop}(3-1)) \\ \longmapsto & \#(\mathcal{A}_s\, \mathit{loop}(2)) \\ \longmapsto & \#(\mathcal{A}_s\, \mathcal{K}(\lambda k'.\mathcal{A}_s\, \mathit{loop}(2-1))) \\ \longmapsto & \#(\mathcal{A}_s\, (\mathcal{A}_s\, \mathit{loop}(2-1))) \\ \longmapsto & \#(\mathcal{A}_s\, (\mathcal{A}_s\, \mathit{loop}(1))) \\ \longmapsto & \#(\mathcal{A}_s\, (\mathcal{A}_s\, 1)) \\ \longmapsto & \#(1) \\ \longmapsto & 1 \end{array}$$

Notice how we develop a stack of pending $\mathcal{A}_s$ operations above: $\#(\mathcal{A}_s\, (\mathcal{A}_s\, \ldots)$. The size of this stack is proportional to the value of the *loop* argument. In other words, we have the same space leak using the L_{AK} encoding here, as we did with the encoding of L_{cc} into L_{CP} in the previous chapter. Our improved encoding presented in the next section addresses this problem.

Using the traditional encoding of L_{SR} into L_{AK}, we can also observe unfaithfulness by using effects in a way similar to the example in the previous chapter. Consider the example:

$$\#(\mathcal{S}(\lambda k.\texttt{raise Fail})\ \texttt{handle Fail} \Rightarrow 99)\ \texttt{handle Fail} \Rightarrow 0$$

and its reduction sequence in L_{SR}:

$$\begin{array}{ll} & \#(\mathcal{S}(\lambda k.\texttt{raise Fail})\ \texttt{handle Fail} \Rightarrow 99)\ \texttt{handle Fail} \Rightarrow 0 \\ \longmapsto & \#(\texttt{raise Fail})\ \texttt{handle Fail} \Rightarrow 0 \\ \longmapsto & (\texttt{raise Fail})\ \texttt{handle Fail} \Rightarrow 0 \\ \longmapsto & 0 \end{array}$$

Instead, running the encoded version:

$$\#(\mathcal{K}(\lambda k.\mathcal{A}_s\ (\texttt{raise}\,\texttt{Fail}))\ \texttt{handle}\,\texttt{Fail} \Rightarrow 99)\ \texttt{handle}\,\texttt{Fail} \Rightarrow 0$$

results in the following sequence:

$$\begin{array}{ll} & \#(\mathcal{K}(\lambda k.\mathcal{A}_s\ (\texttt{raise}\,\texttt{Fail}))\ \texttt{handle}\,\texttt{Fail} \Rightarrow 99)\ \texttt{handle}\,\texttt{Fail} \Rightarrow 0 \\ \longmapsto & \#((\mathcal{A}_s\ (\texttt{raise}\,\texttt{Fail}))\ \texttt{handle}\,\texttt{Fail} \Rightarrow 99)\ \texttt{handle}\,\texttt{Fail} \Rightarrow 0 \\ \longmapsto & \#(99)\ \texttt{handle}\,\texttt{Fail} \Rightarrow 0 \\ \longmapsto & 99\ \texttt{handle}\,\texttt{Fail} \Rightarrow 0 \\ \longmapsto & 99 \end{array}$$

Once again, we see that the encoded version performs an action (in this case, raising an exception) in a context that is incorrect with respect to the original semantics. This results in the encoded raised exception *seeing* a different world (set of handlers) than the original raised exception would see. In the next section of this chapter, we will present an improved version of this encoding that does not have these problems.

When we consider the $L_{AK} \rightarrow L_{cc}$ encoding, we do not find any problems with Filinski's version, provided that any expression is preprocessed by surrounding it with a prompt, as suggested by Filinski [11]:

We also need to initialize mk to the initial continuation; the easiest way to do this is to simply wrap a reset around any top-level expression to be evaluated.

What we expected was that if we failed to provide such a surrounding prompt, the encoded version would issue a MissingReset error if a prompt was necessary and not found. Indeed, if the expression $\mathcal{S}(\lambda k.99)$ is evaluated using this encoding, the result is a `MissingReset` error:

$$\begin{array}{ll} & \mathcal{S}(\lambda k.99) \\ = & \texttt{callcc}(\lambda k'.(\mathcal{A}_s\ 99)) \\ \longmapsto & A_s\ 99 \\ \longmapsto & \texttt{MissingReset} \end{array}$$

But a similar program returns the value 99, and does not raise the error:

$$
\begin{array}{ll}
 & \mathcal{S}(\lambda k.k\ 99) \\
= & \texttt{callcc}(\lambda k'.(\mathcal{A}_s\ (k\ 99)[\lambda x.\#(k'\ x)/k])) \\
= & \texttt{callcc}(\lambda k'.(\mathcal{A}_s\ ((\lambda x.\#(k'\ x))\ 99))) \\
\longmapsto & \mathcal{A}_s\ ((\lambda x.\#(k'\ x))\ 99) \\
 & \text{where } k' = \langle \Box \rangle \\
\longmapsto & \mathcal{A}_s\ (\#(k'\ 99)) \\
\longmapsto & \mathcal{A}_s\ (\#(k'\ 99)) \\
\longmapsto & \mathcal{A}_s\ 99 \\
\longmapsto & \text{NOT YET COMPLETE}
\end{array}
$$

However, if the previous encoded program is executed in ML using the Filinski implementation, then executing $\mathcal{S}(\lambda k.99)$ again will result in an ML error:

```
Error:  throw from one top-level expression into another.
```

This means that the use (or misuse) of a $\mathcal{S}$ outside of an enclosing prompt can potentially corrupt the data structure mk, such that subsequent uses of $\mathcal{S}$ exhibit incorrect behavior.

Our contribution is described in the next section, where we present improved versions of both stages of the Filinski encoding; these versions address both the problems with reordering of effects and with the above inconsistency.

6.3 An Improved Encoding of Composable Continuations

We described two stages of the traditional encoding above, and showed problems with each stage. Our research contribution is to provide improved encodings for each of these stages. First, we will present the improved $L_{SR} \rightarrow L_{AK}$ encoding, followed by the improved $L_{AK} \rightarrow L_{cc}$ encoding. Finally, we combine the encodings into a unified $L_{SR} \rightarrow L_{cc}$ encoding, and demonstrate its correctness.

In the case of encoding the composable L_{SR}, we found the same problems of space leak and effect reordering as when we were encoding the abortive L_{cc} into L_{CP}. And our solution is similar: we will encode the body of the $\mathcal{S}$ operator so that it is a thunk, which we can execute at the proper time, after the needed context has been established.

We can concisely specify this transformation as a preprocessing step, where we transform our L_{SR} program into one in which the bodies of the $\mathcal{S}$ operators have been encoded as thunks, and the contexts delimited by $\#$ have been encoded as thunk applications. This will ensure that all $\mathcal{S}$ operations in the preprocessed program have bodies that are thunk values. After a program is preprocessed into thunks, we can proceed with the original Filinski encoding of this into L_{AK}, and then into L_{cc}.

The preprocessing transformation is described formally below; we use a restricted operator $\mathcal{S}_s$ to emphasize the fact that the $\mathcal{S}$ operator is strict:

$$\begin{array}{lcl} \#\, e & \triangleq & (\#(\lambda v.\lambda\_.v)\ e)\ () \\ \mathcal{S}(\lambda k.e) & \triangleq & \mathcal{S}_s(\lambda k'.\lambda\_.(\#\, e[\lambda v.(k'\ v\ ())/k])) \end{array}$$

Notice how the expression e delimited by the prompt has been changed into an expression $(\lambda x.\lambda\_.x)\ e$, which will evaluate e, and use its result value $v \triangleq [\![e]\!]$ to build a thunk $\lambda\_.v$. Under normal circumstances, this thunk would become the result of the prompt, and would then be applied to $()$, resulting in v, which is what we would expect. In the case where e contains a $\mathcal{S}$ operation, the encoding ensures that any invocation of the continuation is passed a thunk, to conform to the encoded shape of the context, which is expecting a thunk.

The following example illustrates the preprocessed version of $\#(1 + \mathcal{S}(\lambda k.k\ 2) + 3)$, and its reduction sequence in L_{SR} (where all occurrences of $\mathcal{S}$ are thunked and replaced with $\mathcal{S}_s$):

$$\begin{array}{ll}
& [\![\#(1+\mathcal{S}(\lambda k.k\ 2)+3)]\!] \\
= & \#((\lambda v.\lambda\_.v)\ \ (1+\mathcal{S}_s(\lambda k'.\lambda\_.\ (\#((\lambda v.(k'\ v\ ()))\ 2))\ )+3)\ () \\
\longmapsto & \#(\lambda\_.(\#((\lambda v.(k'\ v\ ()))\ 2)))\ () \\
& \text{where } k' = \langle\!\langle(\lambda v.\lambda\_.v)\ (1+\Box+3)\rangle\!\rangle. \\
\longmapsto & (\lambda\_.(\#((\lambda v.(k'\ v\ ()))\ 2)))\ () \\
\longmapsto & \#((\lambda v.(k'\ v\ ()))\ 2) \\
\longmapsto & \#(k'\ 2\ ()) \\
= & \#(((\lambda v.\lambda\_.v)\ (1+2+3))\ ()) \\
= & \#(((\lambda v.\lambda\_.v)\ 6)\ ()) \\
\longmapsto & \#(((\lambda\_.6)\ ()) \\
\longmapsto & \#(6) \\
\longmapsto & 6
\end{array}$$

6.4 A Proof of Correctness

The final step in establishing our improved encoding is to prove a theorem showing that for any program $P \in L_{SR}$, it will produce the same result as its encoding $[\![P]\!] \in L_{cc}$. As in the previous chapter, we also wish to know that an encoding diverges (does not halt) if its source expression diverges, and that if an encoding diverges, then its source diverges. We use the notation $c \Uparrow$ to indicate that there is no value v such that $c \longmapsto\!\!\!\!\rightarrow v$.

Because an encoding may take a different number of steps to reach an answer, and because the expression in the encoding is not identical with the source expression, we define a *simulation* relation between source configurations and encoded configurations. This relation is indicated as $\lceil c \rceil \sim c$, which means that the encoding of c, called $\lceil c \rceil$, simulates c.

This simulation relation is defined inductively on the structure of the source language. We can prove a theorem which establishes that for each configuration c we have in a source reduction sequence, there is a configuration $\lceil c \rceil$ in the encoded configuration's reduction sequence that simulates c, and vice versa.

Theorem 6.4.1. *Given source configuration c, the following properties hold:*

1. $\forall c'.c \longmapsto\!\!\!\!\!\rightarrow c' \not\longmapsto \Rightarrow \exists c''.\lceil c \rceil \longmapsto\!\!\!\!\!\rightarrow c'' \wedge c'' \sim c'$

2. $\forall c''.\lceil c \rceil \longmapsto\!\!\!\!\!\rightarrow c'' \not\longmapsto \Rightarrow \exists c'.c \longmapsto\!\!\!\!\!\rightarrow c' \wedge c'' \sim c'$

3. $c \Uparrow \iff \lceil c \rceil \Uparrow$

6.5 Summary

We were able to build an encoding that faithfully preserves the order of evaluation of the original expression. We did this by removing an implicit evaluation that was being forced by a strict abortive continuation, and replacing it with a thunk that allowed us to explicitly force the execution to occur after the context was aborted and the new context established.

The appendix to this report contains Standard ML implementations of both the original and the improved versions of these operators, as well as implementations of the above test cases.

CHAPTER VII

Conclusion

We have presented a family of functional programming languages, L_0, L_{CP}, L_{SR}, and others, each of which features a particular suite of control operators. We have shown how to encode the operators of a source language into expressions in a target language containing different operators. The primary encodings we described are the encoding of a the undelimited, abortive language L_{cc} into the undelimited, abortive language L_{C_u}, and the encoding of the delimted, composable L_{SR} into the undelimited, abortive language L_{cc} (with a single mutable variable).

Our contribution includes test programs that highlight errors in these encodings, improved versions of these encodings, and a characterization of the systematic nature of these errors. We discovered that operators in the source language that are not strict must be handled with care during the encoding. If a non-strict operator is encoded using a target language expression that is strict, then it is important to be aware of when the expression is collapsed to a value, possibly using delay/force to control the order of evaluation.

The presence of effects that are context-dependent (e.g., exceptions, I/O, state) requires that we are careful about controlling this collapse and ensuring that it occurs in the intended context. In both of our improved encodings, we do this by abstracting the expression as a function (a thunk), which enables us to execute the expression (collapse it to a value) in the desired context.

APPENDIX I

Reading List

Listed below are the sources that have served as the core material for my research. I have distinguished between the general background material common to most programming language research, and the specific articles related to the subject of control operators.

1.1 Programming Languages Background

- *Theories of Programming Languages* [22] (1998) by John C. Reynolds is a textbook describing many of the techniques and principles underlying programming language research. I have focused on the following chapters: 1, 2, 5, 6, 10, 11, 12, 13, 14, and the Appendix. The major emphasis of this book is upon denotational semantics, although the later chapters illustrate how to develop an operational semantics from a denotational semantics.

- *Programming Languages and Lambda Calculi (draft)* [9] (2003) by Matthias Felleisen and Matthew Flatt is a book that coherently describes many of the formal methods and results used in programming language research. Specifically, the book develops the lambda calculus and refines Landin's ISWIM model of an abstract language and machine. It rigorously uses formal translations and proofs of simulation to describe execution of a variety of programming language features such as exceptions, continuations, storage, and control operators in general. This book emphasizes operational semantics.

- *A Structural Approach to Operational Semantics* [19] (1981) by Gordon R. Plotkin is a collection of notes that have served to introduce Structured Operational Semantics to the community and are still very relevant.

- *On the Expressive Power of Programming Languages*[8] (1991) by Matthias Felleisen defines various notions of expressibility and reviews several encodings.

- *Programming Language Semantics* [24] (1997) by David A. Schmidt is a very readable introduction to denotational semantics. He also provides some interesting perspectives on programming language design and structure.

- *Fundamental Concepts in Programming Languages* [28] (1967) by Christopher Strachey.

- *Call-by-Name, Call-by-Value and the Lambda-Calculus* [18] (1975) by Gordon R. Plotkin.

- *Definitional Interpreters for Higher-Order Programming Languages* [21] (1998) by John C. Reynolds.

- *From Language Concepts to Implementation Concepts* [16] (2000) by Robert Milne.

1.2 Continuations and Control Operators

- *Adding Delimited and Composable Control to a Production Programming Environment* [12] (2000) by Matthew Flatt and Gang Yu and Robert Bruce Findler and Matthias Felleisen. This describes the use of *continuation marks* within Scheme and how this general facility can be used to implement delimited and composable control and a variety of other effects within a programming language. The main contribution that I find is the idea that dynamically scoped variables capture most of the interesting effects that are described in the other papers below.

- *Delimited Dynamic Binding* [15] (2006) by Oleg Kiselyov and Chung-chieh Shan and Amr Sabry. This paper describes the *delimited control* and *dynamic binding* features of programming languages, and then demonstrates that the dynamic binding feature can be macro-expressed in a language containing delimited control. This paper also refers to problems that occur when a control operator is embedded in a language with strict semantics, and provides an example of a fix that resembles ours.

- *Functional Pearl: The Great Escape* [14] (2007) by David Herman describes some of the problems with raw `callcc` and its interactions with exception handling. This paper also provides a useful simulation model and proofs for Filinski's encoding.

- *Abstracting Control* [5] (1990) by Olivier Danvy and Andrzej Filinski. The authors show how various *effects* such as mutable storage, exceptions, threads, and continuations, can all be expressed as by a suitable translation into a *monadic* language framework, which itself can be implemented in a language containing only `shift` and `reset`.

- *Representing Control* [6] (1992) by Olivier Danvy and Andrzej Filinski.

- *Representing Monads* [11] (1994) by Andrzej Filinski. The author reviews the monadic translation as described in *Abstracting Control*, and then provide a translation from `shift` and `reset` into `callcc` and a mutable storage location. This translation is then used to construct an ML implementation of `shift` and `reset`.

- *The Theory and Practice of First-Class Prompts* [7] (1998) by Mattias Felleisen seems to be a good starting point. This paper describes the `control` and `prompt` constructs, which are similar in purpose to `shift` and `reset`.

- *The Discoveries of Continuations* [20] (1993) by John C. Reynolds.

- *The Revised Report on the Syntactic Theories of Sequential Control and State* [10] (1992) by Matthias Felleisen and Robert Hieb. This is the canonical description of the control and prompt operators, and is cited by the Delimited Dynamic Binding paper for its use of a trampoline to avoid the premature by-value conversion.

- *Reasoning about Programs in Continuation-Passing Style* [23] (1993) by Amr Sabry and Matthias Felleisen.

- *Continuations Revisited* [31] (2000) by Christopher P. Wadsworth.
- *Continuations: A Mathematical Semantics for Handling Full Jumps* [29] (2000) by Christopher Strachey and Christopher P. Wadsworth.

1.3 Historical and Biographical

- *Christopher Strachey and Fundamental Concepts* [27] (2000) by Joseph E. Stoy.
- *Induction, Domains, Calculi: Strachey's Contributions to Programming-Language Engineering* [25] (2000) by David A. Schmidt.
- *Christopher Strachey - Understanding Programming Languages* [4] (2000) by Rod Burstall.
- *A Foreword to 'Fundamental Concepts in Programming Languages'* [17] (2000) by Peter D. Mosses.
- *Some Reflections on Strachey and His Work* [26] (2000) by Dana Scott.

APPENDIX II

ML Implementations and Examples

SML/NJ Implementations

Listing II.1 contains Filinski's original ML implementation of `escape`, which allows for the capture of undelimited continations (via `SMLofNJ.callcc`) and translates the native `SMLofNJ.Cont.cont` into an invokable function. Both Filinski's and our improved encoding use this implementation of `escape`.

Listing II.1: escape.sml

```
signature ESCAPE = sig
    type void
    val coerce : void -> 'a
    val escape : ( ( '1a -> void ) -> '1a ) -> '1a
end

structure Escape : ESCAPE = struct
local
    val callcc = SMLofNJ.Cont.callcc
    val throw = SMLofNJ.Cont.throw
in
    datatype void = VOID of void
    fun coerce (VOID v) = coerce v

    fun escape f = callcc( fn k => f ( fn x => throw k x ) )
end
end
```

Listing II.2 contains several functors which implement the CONTROL signature. We have extended this signature with the $\mathcal{C}$ and $\mathcal{A}$ operators, to reduce the amount of source files we needed to maintain.

Listing II.2: control.sml

```
use "escape.sml";

(* Filinski's CONTROL extended with initialize, abort and C *)

signature CONTROL = sig
    type ans
    val initialize : ( unit -> unit );
    val abort : ( unit -> ans ) -> '1a
    val C : ( ( '1a -> ans ) -> ans ) -> '1a
    val reset : ( unit -> ans ) -> ans
    val shift : ( ( '1a -> ans ) -> ans ) -> '1a
end


(* Filinski's Control extended with initialize, abort and C *)

functor Control( type ans ) : CONTROL = struct
local
    open Escape
    exception MissingReset
in
    val mkInitial : ( ans -> void) = fn _ => raise MissingReset;
    val mk = ref( mkInitial );

    fun initialize () = ( mk := mkInitial );

    fun abort  x = coerce( !mk ( x () ) ) (* Incorrect semantics *)
    fun abortV x = coerce( !mk x )

    type ans = ans

    fun reset t =
        escape( fn k =>
                let
                    val m = !mk
                in
                    mk := ( fn r =>
                            ( mk := m; k r ) );
                    abortV( t () )
                end )
    fun shift h =
        escape( fn k =>
                abortV( h ( fn v =>
                            reset( fn () =>
                                        coerce( k v ) ) ) ) )
    fun C h =
        escape( fn k =>
                abortV( h ( fn v =>
                            coerce( k v ) ) ) )
```

```
end
end


(*
 *  NewControl contains versions of shift and reset that assume
 *  that the abort() deletes the prompt, which must be compensated
 *  for by inserting a new prompt into the metacontinuation.
 *  This is the version that corresponds to the LPAR submission.
 *)

functor NewControl( type ans ) : CONTROL = struct
local
    open Escape
    exception MissingReset
in

    val mkInitial : ( ( unit -> ans ) -> void ) =
            fn r => ( raise MissingReset );
    val mk = ref( mkInitial );

    fun initialize () = ( mk := mkInitial );
    fun abort thunk = coerce( !mk thunk )

    type ans = ans

    fun reset t =
        escape( fn k =>
                let
                    val m = !mk
                in
                    mk := ( fn x =>
                            ( mk := m; k x ) );

                    let
                        val result = ( t () )
                    in
                        abort( fn () => result )
                    end
                end ) ()

    fun shift h =
        escape( fn k =>
                abort( fn () =>
                       reset ( fn () =>
                               h ( fn v =>
                                   reset( fn () => coerce( k v ) )
                                 )
                             )
                     )
                )

    fun C h =
        escape( fn k =>
                abort( fn () =>
                       reset ( fn () =>
                               h ( fn v => coerce( k v ) )
                             )
                     )
              )
end
end


(*
 *  This version of control assumes that abort does
 *  not delete the meta-continuation until after
 *  successfully executing it. This obviates the need to
 *  insert an extra prompt.
 *
 *  This implementation may perhaps be an optimization;
 *  however, it is not proven correct yet.
 *)

functor NewControlOptimized( type ans ) : CONTROL = struct
local
    open Escape
    exception MissingReset
in
    val mkInitial : ( ( unit -> ans ) -> void ) =
            fn r => ( raise MissingReset );
    val mk = ref( mkInitial );

    fun initialize () = ( mk := mkInitial );
    fun abort thunk = coerce( !mk thunk )

    type ans = ans

    fun reset t =
        escape( fn k =>
                let
                    val m = !mk
                in
                    mk := ( fn e =>
                            ( k ( fn () =>
                                   let
                                       val v = ( e () )
                                   in
                                       mk := m;
                                       v
                            end ) ) );
                    let
                        val result = ( t () )
                    in
                        abort( fn () => result )
                    end
                end ) ()

    fun shift h =
        escape( fn k =>
                abort( fn () =>
                       h ( fn v =>
                           reset( fn () => coerce( k v ) )
                         )
                     )
              )

    fun C h =
        escape( fn k =>
                abort( fn () =>
                       h ( fn v => coerce( k v ) )
                     )
              )
end
end
```

Listing **??** contains several functors which implement the CALLCC signature. (Should this be called K, once it is in a reset context?)

Listing II.3: label

```
use "control.sml";

signature CALLCC = sig
    type ans
    val callcc : ( (ans -> ans) -> ans ) -> ans
end


(*
    CallCCNative is a simple wrapper around the SMLofNJ callcc. It
    simply allows the continuation to be invoked, rather than
    requiring a 'throw'
 *)

functor CallCCNative( structure control : CONTROL ) : CALLCC =
    struct
local
    open control
in
```

```
    type ans = ans

    fun callcc h =
        SMLofNJ.Cont.callcc( fn k =>
            h ( fn v => SMLofNJ.Cont.throw k v ) )
end
end


(*
    CallCCOldC is the 'traditional' implementation of callc
    on top of C
 *)

functor CallCCOldC( structure control : CONTROL ) : CALLCC = struct
local
    open control
in
    type ans = ans

    fun callcc h =
        C( fn ( k' ) =>
            let
                val result = h ( fn v => k' v )
            in
                k' result
            end )
end
end


(*
    CallCCNewC is the New implementation of callc on top of C that
    uses thunking to delay execution until after the jump.

    Note that the implementation below uses beta-reduction via
    application as the means to perform the substitution demanded by
    the encoding in the paper.

    To perform an actual substitution would require a fully-
        reflective
    language like Scheme, where the parameter to callcc could be
    decomposed and edited.

    One of the questions that Zena has posed is whether this
    implementation via beta-reduction is equivalent to the
    substitution-based encoding in the paper on page 6.
 *)

functor CallCCNewC( structure control : CONTROL ) : CALLCC = struct
```

```
local
    open control
in
    type ans = ans

    fun callcc h =
            C( fn ( k' ) =>
                k' ( fn _ =>
                    h ( fn x =>
                        ( k' ( fn _ => x ) )
                      )
                  )
             ) ()
end
end
```

SML/NJ Examples

Composable Continuation Usable Outside of Reset

As referenced in the chapter on composable continuations, Figure II.4 is the example we used to confirm to ourselves that a composable continuation is safely usable outside of any delimiter. Note that the union type in the example is needed to allow the expression to typecheck in ML.

Listing II.4: ShiftedContUsedOutsideReset

```
use "control.sml";

local
    datatype    bundle = Val of int | Cont of ( int -> bundle );
    structure   F = Control( type ans = bundle );
    structure   N = NewControl( type ans = bundle );
    structure   O = NewControlOptimized( type ans = bundle );
    open        F
in
    val zz =
            let
                val b = reset(
                        fn () => Val( shift(
                                      fn k => Cont k
                                          )
                                    )
                              )
            in
                case b of
                    Cont kk =>
```

```
                                        (
                                            print "[1]\n";
                                            kk 0
                                        )
                            |
                            Val v =>
                                        (
                                            print "[2]\n";
                                            Val v
                                        )
                    end
        end;
```

APPENDIX III

Extended Continuation Passing Style

In the chapter *Composable Continuations*, we made the case that composable continuations act like normal functions, and that they can be invoked outside of any delimiter. In this appendix, we encode an example into the original ECPS semantics of Danvy and Filinski.

First we must translate the expression into the meta-continuation semantics by using the denotational semantics in [11] and reproduced in 3.1. We take this translation and apply it to our program $P = (\#\mathcal{S}(\lambda k.k))\ 0$ in Figure 3.2.

To run program P, evaluate:

$\mathcal{E}[\![P]\!]\ \rho_0\ \kappa_0\ \gamma_0$

where ρ_0 is the initial store

where κ_0 is the initial continuation, $\lambda x.\lambda\gamma.\gamma x$

where γ_0 is the initial metacontinuation, usually $\lambda x.x$

$$\mathcal{E}[\![x]\!]\ \rho = \lambda\kappa.\lambda\gamma.\kappa\ (\rho x)\ \gamma$$

$$\mathcal{E}[\![\lambda x.E]\!]\ \rho = \lambda\kappa.\lambda\gamma.\kappa(\lambda v.\lambda\kappa'.\lambda\gamma'.\ \mathcal{E}[\![E]\!](\rho[x \mapsto v])\ \kappa'\ \gamma')\ \gamma$$

$$\mathcal{E}[\![E_1 E_2]\!]\ \rho = \lambda\kappa.\lambda\gamma.\mathcal{E}[\![E_1]\!]\ \rho(\lambda f.\lambda\gamma'.\mathcal{E}[\![E_2]\!]\ \rho(\lambda a.\lambda\gamma''.f\ a\ \kappa\ \gamma'')\ \gamma')\ \gamma$$

$$\mathcal{E}[\![\mathcal{K}E]\!]\ \rho = \lambda\kappa.\lambda\gamma.\mathcal{E}[\![E]\!]\ \rho(\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.\kappa\ v\ \gamma'']\ \kappa\gamma')\ \gamma$$

$$\mathcal{E}[\![\mathcal{S}E]\!]\ \rho = \lambda\kappa.\lambda\gamma.\mathcal{E}[\![E]\!]\ \rho(\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.\kappa\ v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\ \gamma')\ \gamma$$

$$\mathcal{E}[\![\#E]\!]\ \rho = \lambda\kappa.\lambda\gamma.\mathcal{E}[\![E]\!]\ \rho\ (\lambda x.\lambda\gamma'.\gamma' x)\ (\lambda r.\kappa\ r\ \gamma)$$

FIGURE 3.1: This is the meta-continuation semantics for a language containing the shift and reset operators. This is the version described in Danvy's and Filinksi's *Representing Monads*.

Name		*Expression*	
ρ_0	$\triangleq$	$\lambda v.error$	Initial environment
κ_0	$\triangleq$	$\lambda x.\lambda\gamma.\gamma\ x$	Initial continuation
γ_0	$\triangleq$	$\lambda x.x$	Initial metacontinuation
P	$\triangleq$	$(\#\texttt{shift}(\lambda k.k))\ 0$	Original program
P_0	$\triangleq$	$\mathcal{E}\llbracket P\rrbracket\ \rho_0\ \kappa_0\ \gamma_0$	Wrap P in an initial execution context
$\mathcal{E}\llbracket P\rrbracket$	$=$	$\mathcal{E}\llbracket(\#\texttt{shift}(\lambda k.k))\ 0\rrbracket$	
	$=$	$\mathcal{E}\llbracket E_1E_2\rrbracket$	
		where $E_1 \triangleq (\#\texttt{shift}(\lambda k.k))$	
		where $E_2 \triangleq 0$	
	$=$	$(\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_1\rrbracket\ \rho(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho(\lambda a.\lambda\gamma''.f\ a\ \kappa\ \gamma'')\ \gamma')\ \gamma)$	
$\mathcal{E}\llbracket E_1\rrbracket$	$=$	$\mathcal{E}\llbracket(\#\texttt{shift}(\lambda k.k))\rrbracket$	
	$=$	$\mathcal{E}\llbracket(\#E_3)\rrbracket$	
		where $E_2 \triangleq E_3 \triangleq \texttt{shift}(\lambda k.k)$	
	$=$	$\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_3\rrbracket\ \rho(\lambda x.\lambda\gamma'.\gamma' x)\ (\lambda r.\kappa\ r\ \gamma)$	
$\mathcal{E}\llbracket E_3\rrbracket$	$=$	$\mathcal{E}\llbracket\texttt{shift}E_4\rrbracket$	
		where $E_4 \triangleq \lambda k.k$	
	$=$	$\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_4\rrbracket\ \rho$	
		$\quad(\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.\kappa\ v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\gamma')\ \gamma$	
$\mathcal{E}\llbracket E_4\rrbracket$	$=$	$\mathcal{E}\llbracket\lambda k.k\rrbracket$	
	$=$	$\lambda\rho.\lambda\kappa.\lambda\gamma.\kappa\ \ (\lambda v.\lambda\kappa'.\lambda\gamma'.\ \mathcal{E}\llbracket k\rrbracket(\rho[k\mapsto v])\ \kappa'\ \gamma')\ \gamma$	
$\mathcal{E}\llbracket k\rrbracket$	$=$	$\lambda\rho.\lambda\kappa.\lambda\gamma.\kappa\ (\rho\ k)\ \gamma$	
$\mathcal{E}\llbracket E_2\rrbracket$	$=$	$\mathcal{E}\llbracket 0\rrbracket$	
	$=$	$\lambda\rho.\lambda\kappa.\lambda\gamma.\kappa\ 0\ \gamma$	

FIGURE 3.2: ECPS metacontinuation interpretation of $(\#\mathcal{S}(\lambda k.k))\ 0$

$$
\begin{array}{lll}
P_0 & \triangleq & \mathcal{E}\llbracket P\rrbracket\ \rho_0\ \kappa_0\ \gamma_0 \\
 & = & \mathcal{E}\llbracket E_1E_2\rrbracket\ \rho_0\ \kappa_0\ \gamma_0 \\
 & = & (\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_1\rrbracket\ \rho(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho(\lambda a.\lambda\gamma''.f\ a\ \kappa\ \gamma'')\ \gamma')\ \gamma) \\
 & & \quad \rho_0\ \kappa_0\ \gamma_0 \\
 & \longmapsto_\beta & \mathcal{E}\llbracket E_1\rrbracket\ \rho_0\ (\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ \gamma_0 \\
 & = & (\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_3\rrbracket\ \rho\ (\lambda x.\lambda\gamma'.\gamma' x)\ (\lambda r.\kappa\ r\ \gamma)) \\
 & & \quad \rho_0\ (\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ \gamma_0 \\
 & \longmapsto_\beta & \mathcal{E}\llbracket E_3\rrbracket\ \rho_0\ (\lambda x.\lambda\gamma'.\gamma' x)\ (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & = & (\lambda\rho.\lambda\kappa.\lambda\gamma.\mathcal{E}\llbracket E_4\rrbracket\ \rho \\
 & & \quad (\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.\kappa\ v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\gamma')\ \gamma) \\
 & & \quad \rho_0\ (\lambda x.\lambda\gamma'.\gamma' x)\ (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & \mathcal{E}\llbracket E_4\rrbracket\ \rho_0 \\
 & & \quad (\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.(\lambda x.\lambda\gamma'.\gamma' x)v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\gamma') \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & = & (\lambda\rho.\lambda\kappa.\lambda\gamma.\kappa\ (\lambda v.\lambda\kappa'.\lambda\gamma'.\ \mathcal{E}\llbracket k\rrbracket(\rho[k\mapsto v])\kappa'\gamma')\gamma) \\
 & & \quad \rho_0 \\
 & & \quad (\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.(\lambda x.\lambda\gamma'.\gamma' x)v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\gamma') \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda f.\lambda\gamma'.f[\ \lambda v.\lambda\kappa'.\lambda\gamma''.(\lambda x.\lambda\gamma'.\gamma' x)v\ (\lambda w.\kappa'\ w\ \gamma'')]\ (\lambda x.\lambda\gamma''.\gamma'' x)\ \gamma') \\
 & & \quad (\lambda v.\lambda\kappa'.\lambda\gamma'.\ \mathcal{E}\llbracket k\rrbracket\ (\rho_0[k\mapsto v])\ \kappa'\ \gamma') \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda v.\lambda\kappa'.\lambda\gamma'.\ \mathcal{E}\llbracket k\rrbracket\ (\rho_0[k\mapsto v])\ \kappa'\ \gamma') \\
 & & \quad \kappa_1 \\
 & & \quad (\lambda x.\lambda\gamma''.\gamma'' x) \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & & \text{where } \kappa_1 \triangleq \lambda v.\lambda\kappa'.\lambda\gamma''.(\lambda x.\lambda\gamma'.\ \gamma' x)\ v\ (\lambda w.\kappa'\ w\ \gamma'') \\
 & \longmapsto_\beta & \mathcal{E}\llbracket k\rrbracket\ (\rho_0[k\mapsto\kappa_1]) \\
 & & \quad (\lambda x.\lambda\gamma''.\gamma'' x) \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda\rho.\lambda\kappa.\lambda\gamma.\kappa\ (\rho\ k)\ \gamma) \\
 & & \quad (\rho_0[k\mapsto\kappa_1]) \\
 & & \quad (\lambda x.\lambda\gamma''.\gamma''\ x) \\
 & & \quad (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda x.\lambda\gamma''.\gamma''\ x)\ \kappa_1\ (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda\gamma''.\gamma''\ \kappa_1)\ (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0) \\
 & \longmapsto_\beta & (\lambda r.(\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ r\ \gamma_0)\ \kappa_1 \\
 & \longmapsto_\beta & (\lambda f.\lambda\gamma'.\mathcal{E}\llbracket E_2\rrbracket\ \rho_0\ (\lambda a.\lambda\gamma''.f\ a\ \kappa_0\ \gamma'')\ \gamma')\ \kappa_1\ \gamma_0
\end{array}
$$

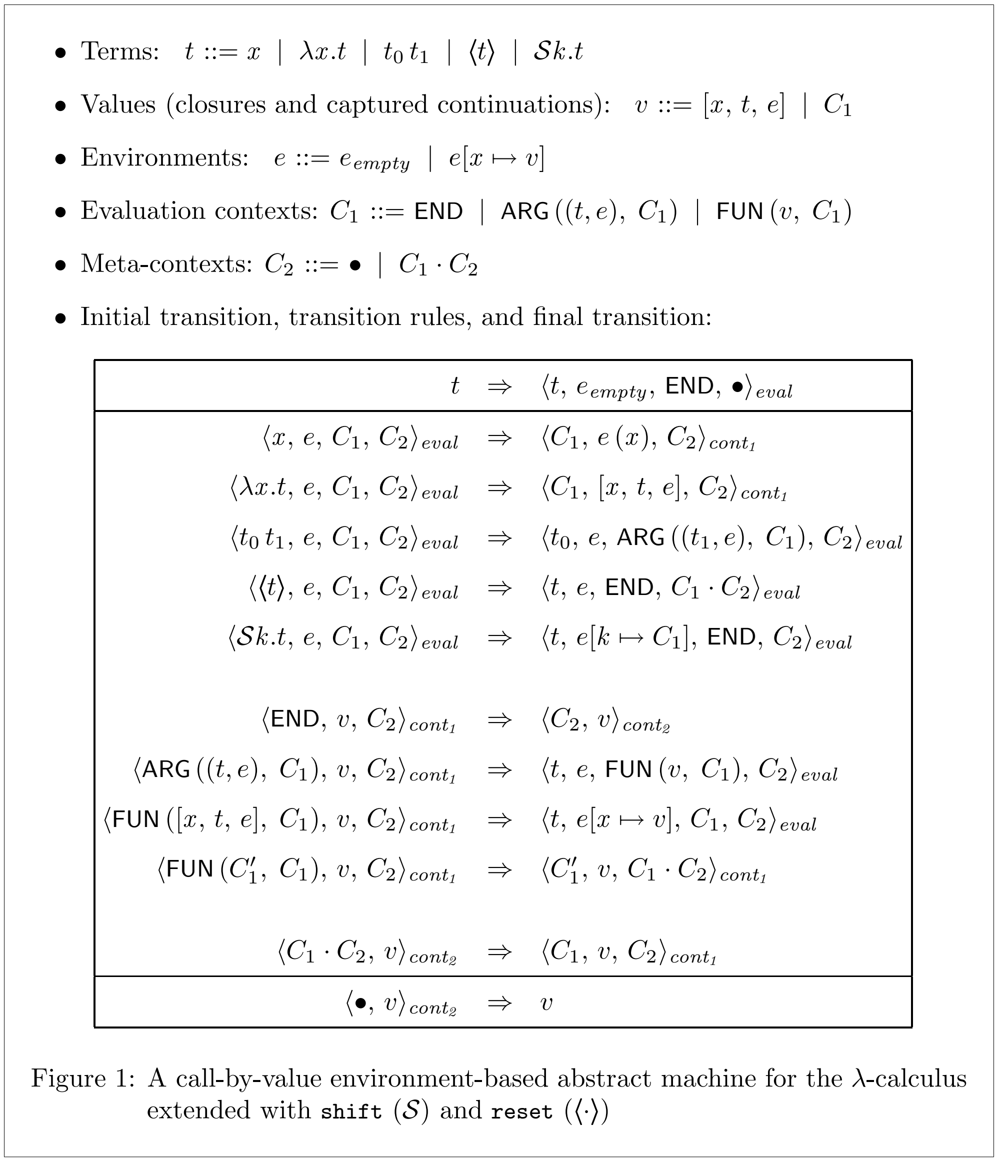
APPENDIX

The `shift/reset` Abstract Machine

In the chapter *Composable Continuations*, we made the case that composable continuations act like normal functions, and that they can be invoked outside of any delimiter. In this appendix, we encode an example into the abstract machine semantics originally defined by Biernacka, Beirnacki, and Danvy [1]. We are actually using the revised version presented in [3, 2]. This machine is presented in Figure 3.5. Using this machine produced the expected result. The detailed evaluation sequence is in the Figure 3.4.

Name		*Expression*	
P	$\triangleq$	$(\#\texttt{shift}(\lambda k.k))\ 0$	Original program
	ι	$\langle\ (\#\texttt{shift}(\lambda k.k))\ 0, \rho_0, END, \bullet\ \rangle_{eval}$	Injected configuration
	$\longmapsto$	$\langle\ (\#\texttt{shift}(\lambda k.k)), \rho_0, ARG((0, \rho_0), END), END \cdot \bullet\ \rangle_{eval}$	
	$=$	$\langle\ \#\texttt{shift}(\lambda k.k), \rho_0, ARG((0, \rho_0), END), END \cdot \bullet\ \rangle_{eval}$	
	$\longmapsto$	$\langle\ \texttt{shift}(\lambda k.k), \rho_0, END, ARG((0, \rho_0), END) \cdot END \cdot \bullet\ \rangle_{eval}$	
	$\longmapsto$	$\langle\ k, \rho_0[k \mapsto END], END, ARG((0, \rho_0), END) \cdot END \cdot \bullet\ \rangle_{eval}$	
	$\longmapsto$	$\langle\ END, \rho_0[k \mapsto END](k), ARG((0, \rho_0), END) \cdot END \cdot \bullet\ \rangle_{cont_1}$	
	$\longmapsto$	$\langle\ ARG((0, \rho_0), END) \cdot END \cdot \bullet, \rho_0[k \mapsto END](k)\ \rangle_{cont_2}$	
	$=$	$\langle\ ARG((0, \rho_0), END) \cdot END \cdot \bullet, END\ \rangle_{cont_2}$	
	$\longmapsto$	$\langle\ ARG((0, \rho_0), END), END, END \cdot \bullet\ \rangle_{cont_1}$	
	$\longmapsto$	$\langle\ 0, \rho_0, FUN(END, END), END \cdot \bullet\ \rangle_{eval}$	
	$\longmapsto$	$\langle\ FUN(END, END), \rho_0(0), END \cdot \bullet\ \rangle_{cont_1}$	
		Above, constants are being treated as names in ρ_0	
	$\longmapsto$	$\langle\ END, 0, END \cdot END \cdot \bullet\ \rangle_{cont_1}$	
	$\longmapsto$	$\langle\ END \cdot END \cdot \bullet, 0\ \rangle_{cont_2}$	
	$\longmapsto$	$\langle\ END, 0, END \cdot \bullet\ \rangle_{cont_1}$	
	$\longmapsto$	$\langle\ END \cdot \bullet, 0\ \rangle_{cont_2}$	
	$\longmapsto$	$\langle\ END, 0, \bullet\ \rangle_{cont_1}$	
	$\longmapsto$	$\langle\ \bullet, 0\ \rangle_{cont_2}$	
	$\longmapsto$	0	Final result

FIGURE 3.4: Abstract Machine Interpretation of $(\#\mathcal{S}(\lambda k.k))\ 0$



- Terms: $t ::= x \mid \lambda x.t \mid t_0\, t_1 \mid \langle t \rangle \mid \mathcal{S}k.t$
- Values (closures and captured continuations): $v ::= [x,\, t,\, e] \mid C_1$
- Environments: $e ::= e_{empty} \mid e[x \mapsto v]$
- Evaluation contexts: $C_1 ::= \mathsf{END} \mid \mathsf{ARG}\,((t,e),\ C_1) \mid \mathsf{FUN}\,(v,\ C_1)$
- Meta-contexts: $C_2 ::= \bullet \mid C_1 \cdot C_2$
- Initial transition, transition rules, and final transition:

t	$\Rightarrow$	$\langle t,\, e_{empty},\, \mathsf{END},\, \bullet \rangle_{eval}$
$\langle x,\, e,\, C_1,\, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, e\,(x),\, C_2 \rangle_{cont_1}$
$\langle \lambda x.t,\, e,\, C_1,\, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle C_1,\, [x,\, t,\, e],\, C_2 \rangle_{cont_1}$
$\langle t_0\, t_1,\, e,\, C_1,\, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t_0,\, e,\, \mathsf{ARG}\,((t_1, e),\ C_1),\, C_2 \rangle_{eval}$
$\langle \langle t \rangle,\, e,\, C_1,\, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t,\, e,\, \mathsf{END},\, C_1 \cdot C_2 \rangle_{eval}$
$\langle \mathcal{S}k.t,\, e,\, C_1,\, C_2 \rangle_{eval}$	$\Rightarrow$	$\langle t,\, e[k \mapsto C_1],\, \mathsf{END},\, C_2 \rangle_{eval}$
$\langle \mathsf{END},\, v,\, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle C_2,\, v \rangle_{cont_2}$
$\langle \mathsf{ARG}\,((t, e),\ C_1),\, v,\, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle t,\, e,\, \mathsf{FUN}\,(v,\ C_1),\, C_2 \rangle_{eval}$
$\langle \mathsf{FUN}\,([x,\, t,\, e],\ C_1),\, v,\, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle t,\, e[x \mapsto v],\, C_1,\, C_2 \rangle_{eval}$
$\langle \mathsf{FUN}\,(C_1',\ C_1),\, v,\, C_2 \rangle_{cont_1}$	$\Rightarrow$	$\langle C_1',\, v,\, C_1 \cdot C_2 \rangle_{cont_1}$
$\langle C_1 \cdot C_2,\, v \rangle_{cont_2}$	$\Rightarrow$	$\langle C_1,\, v,\, C_2 \rangle_{cont_1}$
$\langle \bullet,\, v \rangle_{cont_2}$	$\Rightarrow$	v

Figure 1: A call-by-value environment-based abstract machine for the λ-calculus extended with `shift` ($\mathcal{S}$) and `reset` ($\langle \cdot \rangle$)

FIGURE 3.5: Abstract machine for shift/reset from Biernacki, Danvy, and Shan. TYPESET THIS IN LATEX AND REPLACE e WITH ρ. ADD IN SUPPORT FOR CONSTANTS.

BIBLIOGRAPHY

[1] M. Biernacka, D. Biernacki, and O. Danvy. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *CoRR*, abs/cs/0508048, 2005.

[2] D. Biernacki and O. Danvy. Theoretical Pearl: A simple proof of a folklore theorem about delimited control. *J. Funct. Program.*, 16(3):269–280, 2006.

[3] D. Biernacki, O. Danvy, and C. chieh Shan. On the static and dynamic extents of delimited continuations. *Sci. Comput. Program.*, 60(3):274–297, 2006.

[4] R. M. Burstall. Christopher Strachey - Understanding Programming Languages. *Higher-Order and Symbolic Computation*, 13(1/2):51–55, 2000.

[5] O. Danvy and A. Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 151–160, New York, NY, 1990. ACM.

[6] O. Danvy and A. Filinski. Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[7] M. Felleisen. The theory and practice of first-class prompts. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190, New York, NY, USA, 1988. ACM.

[8] M. Felleisen. On the Expressive Power of Programming Languages. *Sci. Comput. Program.*, 17(1-3):35–75, 1991.

[9] M. Felleisen and M. Flatt. Programming Languages and Lambda Calculi (draft). August 2003.

[10] M. Felleisen and R. Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.*, 103(2):235–271, 1992.

[11] A. Filinski. Representing Monads. In *Conf. Record 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'94, Portland, OR, USA, 17–21 Jan. 1994*, pages 446–457. ACM Press, New York, 1994.

[12] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding Delimited and Composable Control to a Production Programming Environment. In *ICFP*, pages 165–176, 2007.

[13] R. Harper. *Practical Foundations for Programming Languages (draft)*. 2008.

[14] D. Herman. Functional Pearl: The Great Escape or, How to jump the border without getting caught. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 157–164, New York, NY, USA, 2007. ACM.

[15] O. Kiselyov, C. chieh Shan, and A. Sabry. Delimited dynamic binding. In J. H. Reppy and J. L. Lawall, editors, *ICFP*, pages 26–37. ACM, 2006.

[16] R. Milne. From Language Concepts to Implementation Concepts. *Higher-Order and Symbolic Computation*, 13(1/2):77–81, 2000.

[17] P. D. Mosses. A Foreword to 'Fundamental Concepts in Programming Languages'. *Higher-Order and Symbolic Computation*, 13(1/2):7–9, 2000.

[18] G. D. Plotkin. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[19] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[20] J. C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[21] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[22] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[23] A. Sabry and M. Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[24] D. A. Schmidt. Programming Language Semantics. In *The Computer Science and Engineering Handbook*, pages 2237–2254. 1997.

[25] D. A. Schmidt. Induction, Domains, Calculi: Strachey's Contributions to Programming-Language Engineering. *Higher-Order and Symbolic Computation*, 13(1/2):89–101, 2000.

[26] D. S. Scott. Some Reflections on Strachey and His Work. *Higher-Order and Symbolic Computation*, 13(1/2):103–114, 2000.

[27] J. E. Stoy. Christopher Strachey and Fundamental Concepts. *Higher-Order and Symbolic Computation*, 13(1/2):115–117, 2000.

[28] C. Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[29] C. Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000.

[30] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[31] C. P. Wadsworth. Continuations Revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.