NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
DEPARTMENT OF MATHEMATICS

Non-Strict Pattern Matching and Delimited Control

By
Petros Barbagiannis

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Supervised by
Nikolaos Papaspyrou

Thesis Committee
Nikolaos Papaspyrou
Panos Rondogiannis
Ioannis Smaragdakis

Graduate Program in Logic
and Theory of Algorithms and Computation

$\mu\prod\lambda\forall$

October 2017

Η παρούσα Διπλωματική Εργασία

εκπονήθηκε στα πλαίσια των σπουδών

για την απόκτηση του

Μεταπτυχιακού Διπλώματος Ειδίκευσης

στη

Λογική και Θεωρία Αλγορίθμων και Υπολογισμού

που απονέμει το

Τμήμα Μαθηματικών

του

Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών

Εγκρίθηκε την 31/10/2017 από Εξεταστική Επιτροπή

αποτελούμενη από τους:

Ονοματεπώνυμο	Βαθμίδα	Υπογραφή
1. Νικόλαος Παπασπύρου	Αναπληρωτής Καθηγητής	……………………
2. Πάνος Ροντογιάννης	Καθηγητής	……………………
3. Ιωάννης Σμαραγδάκης	Καθηγητής	……………………

To my mother and to the memory of my father

Abstract

It has been long known that continuations and evaluation strategies are two intimately related concepts of functional programming languages. In one of the earliest results, continuation-passing style (CPS) was introduced as a means to decouple the evaluation order of a language from the evaluation order of its interpreter. Since then, this style of programming has been proved extremely useful in areas ranging from compiler implementation to denotational semantics.

Since the introduction of CPS, a wide variety of control operators have been developed. Delimited control operators, in particular, are a powerful mechanism of functional programming languages that generalize traditional first-class control operators, such as `call/cc`, and provide the means to abstract control. One notable application of delimited control operators is the construction of a novel abstract machine for the call-by-need λ-calculus that simulates store-based effects with delimited continuations.

Pattern matching on algebraic data types is an essential feature of functional programming languages. However, pattern matching is often thought to be syntactic sugar that can be merely represented by a proper encoding. In this thesis we study the operational characteristics of non-strict pattern matching. We also explore the semantics of control operators, as well as some of their applications. Finally, we seek to examine the connection between implementing a non-strict pattern matching evaluator and delimited continuations.

Acknowledgements

First and foremost, I wish to thank the MPLA committee members who accepted me in the program, and thus gave me the opportunity to study alongside many highly gifted and motivated people — students and teachers alike.

I would like to thank my advisor Nikolaos Papaspyrou for trusting me and offering me his guidance, as well as the rest of the thesis committee members, Panos Rondogiannis and Ioannis Smaragdakis for their helpful and insightful comments. The courses Type Systems for Programming Languages, taught by N. Papaspyrou, and Semantics of Programming Languages, taught by P. Rondogiannis helped me to discover an extremely fascinating area of study, and changed the way I think about programming languages and software in general.

Last, but not least, I want to express my deepest gratitude to Georgios Fourtounis. Without his suggestions, feedback, support, and guidance this work would have never been possible.

Contents

Chapter 1

Introduction

1.1 Programming Models

The importance of programming languages in modern applications is unquestionable, and their design and implementation has been an incessantly active area of research since the advent of computers. Not all programming languages, however, are designed based on the same model. Some have been heavily influenced, if not originally inspired, from a primitive model of computation — the λ-calculus. Languages falling into this category are called functional programming languages. In this thesis we aspire to comprehend and establish certain aspects of these languages.[1]

A program written in a *purely functional* or *applicative* language consists of expressions which may be composed of sub-expressions (constants, variables, functions, etc.). Evaluating these expressions returns a result — a *value*. In this programming paradigm the notions of function and function application are fundamental (hence the name functional). New functions can be defined in terms of expressions and these functions can in turn be composed with other functions to build programs. In addition, functions can be saved in variables just like any other value, they can be passed to other functions as arguments or returned as results. That is, functions are *first-class* objects. Languages in which functions

[1]This introductory section is not intended to be a complete description of functional programming. For a short, yet extensive, introduction, the interested reader is encouraged to read Hudak's excellent article [16].

enjoy first-class status are said to be *higher-order*.

In a purely functional program we can substitute an expression for any other, whose value is the same, without altering the result of the original program. More precisely, when an expression is used in a certain context, the value of the expression is all that matters, and if we replace it with another equivalent — in terms of its value — expression, the meaning of the context will not be affected. This property, which has its origins in mathematics, is known as *referential transparency* [25]. In a referentially transparent programming language, functions seem and behave more like mathematical objects rather than just programming constructs, and helps us reason about our programs; a task that otherwise would have been much harder.

By contrast, an *imperative* language contains constructs, called *statements* or *commands*, that differ from expressions in that the former are used to carry out actions that may change, or *mutate*, the program's state. Thus, statements are said to have *side effects*. For example, the *assignment* statement assigns a value or changes an existing value of a variable. When a mutating assignment statement (e.g., `x := x + 1`) is introduced inside the body of a function — which is not allowed in a purely functional program — calling this function twice yields different results. This means that the value of the function depends on the value of the variable `x` which is different in every function call. Consequently, we can no longer deem that referential transparency holds for this function.

1.2 Semantics

A programming language is identified by its *syntax* and its *semantics*. The syntax of a language consists of the grammar rules that specify how a program, written in this language, should be structured. Nevertheless, being well-structured (or syntactically correct) is necessary but not sufficient for a program to be entirely correct. Semantics fills this gap and helps us understand a program's behaviour by giving it an interpretation — its *meaning*.

In his landmark book, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* [25], Joseph Stoy remarks that a formal semantics can serve a threefold purpose: (a) provide compiler implementers with a precise definition of the language, (b) help programmers understand and reason about the behaviour of their programs and (c) guide language designers to improve programming languages. In other words, semantics is an indispensable tool that allows us to understand and describe all aspects of a programming language with rigor and accuracy.

In this work, we mainly focus on the second and third role. More precisely, we use the formal semantics of specific language constructs to study their behaviour and thereby be able to put these constructs together in a way that expands the language's expressiveness.

There are various approaches to defining a semantics. Among them, is the *operational* and the *denotational* approach. The first method, as its name implies, is concerned with properties of the execution of a program. The operational semantics often takes the form of an *abstract machine*, i.e., a prototype implementation (or an interpreter) of the language. This type of machine aims at defining the behaviour of a language. In doing so, it abstracts away low-level details, such as the concrete syntax of the language, and permits us to concentrate on more general, yet essential things such as the result of a computation and the execution steps that are followed.

An *abstract machine* consists of a set of transition rules, where on each step, a transition is chosen according to the state of the machine's input. Typically, this state is the abstract syntax of the expression at a specific moment but, as we will see, it can also include other parameters such as the surrounding context. Given a program, its meaning is taken to be the exact sequence of transitions that were followed during the program's execution by the abstract machine. The last element of this sequence is the final state of the machine and it denotes the value computed by the program. While an abstract machine hides many of the details, whose importance is sometimes not inconsiderable, it can act as a high-level specification, based on which a lower-level implementation can be built.

Denotational (or mathematical) semantics is another method of formal semantics where the meaning of a program is given by a *semantic function* that maps programs to their *denotations*, i.e., mathematical objects, such as numbers, functions or truth values. These objects constitute a *semantic domain* whose elements represent values of programs. Thus, defining a denotational semantics for a language boils down to defining its semantic domain along with a mapping from syntactic constructs of the language to elements of the domain. Note, however, that when defining the denotational semantics of a language, we do not care for the actual execution of a program. In other words, while with the operational approach we are interested in *how* a program terminates (if at all), in denotational semantics we focus on *what* the final result of the program is. The value that is assigned by the semantic function to a program depends on the value that the same function assigns to the expressions of which the program consists. This principle, which is essential to denotational semantics, is called *compositionality.*

It is important to emphasize that different semantic approaches are not mutually exclusive, but rather, complementary to each other. In fact, combining these approaches to demonstrate safety, correctness or any other important property is often used as part of the meta-theory of a well-designed language. Furthermore, showing that these approaches agree with each other is often a challenging task.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows: in Chapter 2 we introduce the features and tools that constitute the building blocks of this work. In Section 2.1 we give a brief introduction of the λ-calculus and its fundamental elements that will assist us in expressing the main ideas studied in the chapters that follow. In Section 2.2 we explore the concept of evaluation strategies which is essential to all programming languages. We also present and compare the most widely used strategies and we concentrate on the advantages of call-by-need (or lazy evaluation). In Sections 2.3 and 2.4 we cover two intimately related features of functional programming languages: algebraic data types and pattern matching. We describe the behaviour of these features in languages with non-strict semantics and we attempt to uncover their operational characteristics via the use of examples.

Chapter 3 is devoted to continuations. In the introductory section of this chapter we explain the basic concepts. Next, in Section 3.2 we explore continuation-passing style and show how a program written in direct style can be transformed into CPS. Then, we see how CPS can be used to eliminate the evaluation order dependency between a language and its interpreter. In Sections 3.3 and 3.4 we provide an overview of control operators and we consider some of their most notable applications. This chapter ends with Section 3.5 in which we examine the most general form of a control abstraction mechanism:

delimited control operators. In this section we also review some well-known proposals of these operators found in the literature and we study their behaviour.

In Chapter 4, we seek to explore the relationship between non-strict pattern matching and delimited control. In Section 4.1 we provide a detailed description of an abstract machine that is used to evaluate expressions of a call-by-need language. The abstract machine is accompanied with an interpreter that uses delimited continuations to simulate variable bindings. In Section 4.2 we extend the abstract machine so that it also evaluates user-defined types and pattern matching expressions respecting the lazy semantics of the source language. In Section 4.3 we show how the interpreter can be used to simulate our rules.

Chapter 2

Background

This chapter introduces the basic concepts, and lays the foundation for the following chapters. As its title implies, Section 2.1 is concerned with the λ-calculus. In Section 2.2 we explore the fundamental notion of *evaluation strategies*, which is not specific to functional languages, but to every programming language, and we focus on an important concept called *evaluation context*. Next, in Section 2.3 we present a standard abstraction mechanism of many modern functional programming languages called *algebraic data types*, and finally, in Section 2.4 we explain *pattern matching*: a tool that exploits the power of algebraic data types. In this last section we also examine how different evaluation strategies affect the result of a pattern matching expression.

2.1 λ the Ultimate[1]

In the rest of this thesis, we shall be encountering elements of the λ-calculus in a variety of contexts and cases. Even more so, in the last chapter where we explore the call-by-need λ-calculus and we use extensively its basic concepts and notation. It is thereby inevitable to give a presentation of the essentials. Of course, we barely touch on the topic; one can, however, refer to Barendregt's standard text [4] for an in-depth study. A more practical

[1]A phrase that appears in a series of papers, authored by Guy Steele and Gerald Sussman, as well as in the title of a chapter of Daniel Friedman and Matthias Felleisen's classic book *A Little Schemer.* It is also the name of a popular weblog for programming languages.

treatise can be found in Peyton Jones's *The Implementation of Functional Programming Languages* [20] in which it is shown how the λ-calculus can be used as an implementation framework for functional programming languages.

The λ-calculus was invented as a formal system by Alonzo Church in the 1920s and first appeared in a paper published in 1932 [6]. Its original purpose was to provide a foundation for logic, and for mathematics in general, but it was later proved that the system itself was inconsistent. In 1936, Church suggested that the λ-calculus could define the intuitive notion of *effectively calculable functions*, and used it as a formalization to solve negatively the infamous *Entscheidungsproblem*. It was also shown by Kleene and Turing that λ-*definable functions*, *recursive functions* and *Turing-machines* were all three equivalent models that could describe the computational aspects of functions.

Besides computability theory, the λ-calculus has played an important role in the development of functional programming languages, and it has influenced many of their features. Moreover, despite the fact that the λ-calculus is surprisingly simple, it is so expressive that it is often considered as a programming language in its own right.

The syntax of the *pure untyped* λ-calculus is defined by the set of its λ-terms:

$$t ::= x \mid \lambda x.t \mid t\ t$$

where x ranges over an infinite set of variables. Terms of the form $\lambda x.t$ are called *abstractions*, and terms of the form $t\ t$ are called *applications*. The former stand for function definitions while the latter capture the notion of applying a function to an argument, and as we said in Section 1.1, both of these notions are fundamental to programming languages, especially to functional programming languages.

The set FV of a term's *free* variables is defined inductively as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.t_1) &= FV(t_1) - \{x\} \\ FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

An occurrence of a variable x is said to be *bound* in t if $x \notin FV(t)$. In other words, an occurrence of a variable x is bound if it appears in the term t of an abstraction $\lambda x.t$. A term t is *closed* if $FV(t) = \emptyset$.

The basic rewrite rule of the λ-calculus is the *β-axiom* (or *β-reduction*):

$$(\lambda x.t)\ t_1 = t\{x \mapsto t_1\}$$

Here the expression $t\{x \mapsto t_1\}$ denotes the substitution of t_1 for x in t. This rule states that if a term t has the form of an application whose left-hand side term is an abstraction, then t is reduced to another term that consists of the body of the abstraction in which all free occurrences of x are replaced with t_1.

Often, a term may contain more than one sub-term to which the β-axiom can be applied. The evaluation strategy, which is the subject of the next section, determines the order in which the reductions should proceed.

Parentheses are often used to resolve ambiguities such as precedence of operations. When parentheses are omitted we follow two simple conventions. The first convention is that we take application to be left-associative. For example, the term

$$(\lambda x.\lambda y.t_1)\ (\lambda w.t_2)\ (\lambda z.t_3)$$

is equivalent to

$$((\lambda x.\lambda y.t_1)\ (\lambda w.t_2))\ (\lambda z.t_3)$$

The second convention is that abstraction associates to the right. That is, the body of an abstraction extends as far to the right as possible. For example, the term

$$\lambda x.\lambda y.x\ y$$

is equivalent to

$$\lambda x.(\lambda y.(x\ y))$$

In the λ-calculus all functions take a single argument. However, higher-order support — which exists inherently in the λ-calculus — allows multi-argument functions to be thought of as multiple applications of single-argument functions; a process which is known as *currying*. Consider for example that we wish to define a function that takes two arguments and returns a term t. This function can be defined as follows:

$$\lambda x.\lambda y.t$$

The term above is a function that takes an argument, x, and returns another function that takes an argument, y, and returns a term t.

In the next section, we discuss a concept which is related not only to the λ-calculus but to all programming languages: evaluation strategies.

2.2 Evaluation Strategies

As we said in the introduction, a program written in a functional programming language consists of expressions, and computing the result of this program amounts to evaluating the result of each constituent sub-expression. But the order in which these expressions are evaluated is significant and can sometimes affect the end result. Consider for example applying a function `f` to an argument `e`. One option is to first evaluate `e` and pass the returned value to `f`. Another would be to evaluate the body of `f` and if, during its evaluation, `e`'s value is required then (and only then) it is evaluated.

An evaluation strategy is a set of rules that determine which expression (if any) is to be evaluated next in a program. Evaluating this expression — or *redex* as it is called in the context of the λ-calculus — returns a value which in turn may be part of a bigger expression. If evaluating an expression reaches a point where there are no remaining unevaluated sub-expressions, we say that the expression is in *normal form*. An expression whose evaluation does not terminate does not have a normal form.

Yet, there are times that more than one unevaluated sub-expression in an expression is subject to evaluation. Using the example referred to at the beginning of this section, in a function application there are two possible redexes: the argument `e` and the body of the function `f`. An evaluation strategy determines which one of these redexes will be evaluated first. We consider three strategies: *call-by-value* (or *strict*) and *call-by-name* (*non-strict*), as well as *call-by-need* (or *lazy*) as an extension to call-by-name.

call-by-value: In this strategy, the argument is evaluated before the body of the applied function. More precisely, when a function application expression, e.g., `f e`, is evaluated, first `e` is reduced and its value is then substituted for all the occurrences of the formal parameter inside the body of the function. Then, the body of the function is evaluated. As an example, suppose that we define `f` and `e` as follows:

```
f x = x + x
e = 2 + 1
```

The reduction sequence of applying `f` to `e` in a call-by-value language is:

```
f e ⇒ f (2 + 1)
    ⇒ f 3
    ⇒ 3 + 3
    ⇒ 6
```

call-by-name: Unlike call-by-value, the call-by-name strategy proceeds with evaluating the body of the function first, and then evaluates the argument, whenever its value is needed. Note that if the function's formal parameter is not referenced inside the function, then the argument will never be evaluated. We can think of call-by-name as a method that substitutes the argument, intact, for all the occurrences of the formal parameter.[2] It is important to highlight that in the call-by-name strategy the argument is evaluated every time its value is needed. Using `f` and `e` from the previous example, the reduction sequence in a call-by-name language is:

```
f e ⇒ f (2 + 1)
    ⇒ (2 + 1) + (2 + 1)
    ⇒ 3 + (2 + 1)
    ⇒ 3 + 3
    ⇒ 6
```

call-by-need: The call-by-need strategy is a more efficient implementation of call-by-name. Semantically, the two strategies are exactly the same. The difference, however, lies in the former's not re-evaluating the argument once that has already been

[2] Of course, that would be an extremely naive way to follow in an implementation. Abelson and Sussman in [1, Chapter 4] implement an environment-based evaluator. For an even more advanced technique, relevant to non-strict evaluation and based on graph-reduction, the reader can refer to Peyton Jones's [20, Chapter 12].

evaluated. That is, the argument is not evaluated unless it is needed, and once this happens its value is stored (or *memoized*) so that any future reference to it will not have to be re-evaluated. In call-by-need, applying `f` to `e` is almost the same as in call-by-name; but instead of substituting the expression that is bound to `e` for all the occurrences of `x` in `f`, `e` is shared across the entire `f`'s body, so that when it is evaluated once, all subsequent references will point to `e`'s computed value:

$$\begin{aligned} \texttt{f e} &\Rrightarrow \texttt{e + e} \qquad \text{where } \texttt{e} \rightsquigarrow \texttt{2 + 1} \\ &\Rrightarrow \texttt{3 + 3} \\ &\Rrightarrow \texttt{6} \end{aligned}$$

We can think of `e` as a pointer to a location in which the expression `2 + 1` is stored. When `f`'s body is evaluated, all occurrences of the formal parameter are substituted for this pointer. The first time `e`'s value is needed, the expression in that location will be evaluated and the result will be cached. From then on, any reference to `e` will point to the memoized value. This ensures that a single instance of the expression is shared among all occurrences of the formal parameter.

Besides call-by-value's efficiency advantage over call-by-name when the argument is needed more than once, and the opposite when the argument is not needed at all, there is also another difference between the two strategies: if the argument's evaluation does not terminate, that is, it does not have a normal form, then with call-by-name the overall computation is possible to terminate, if the argument's value is not used (or is partly used) inside the function. If, on the other hand, evaluating the argument always terminates, then from a denotational point of view, the semantics of call-by-value and call-by-name give the same result. Call-by-need eliminates the efficiency drawback of call-by-name but retains its advantage of being able to evaluate functions applied to non-terminating arguments that remain unused.

In general, we call a function that will always need the value of its argument *strict*. More formally, a function f is *strict* if:

$$f\ \bot = \bot$$

where $\perp$ (conventionally called *bottom*) denotes non-termination. A language is strict if all of its functions are strict, and non-strict otherwise. The simplest example of a strict function is `id x = x`, or so-called *identity* function. Conversely, an example of a non-strict function is `constant x =` k, where k is a constant. In a non-strict language, applying `constant` to $\perp$ will terminate normally because the argument is never used inside the body of the function. We often use the terms strictness and non-strictness as synonymous with call-by-value and call-by-name (or call-by-need), respectively, but it is important to emphasize that the former is a semantic property whereas the latter is an operational property.

As an example that illustrates strictness vs. non-strictness, consider the following simple function definition:

```
func a b = if a == 1 then 1 else b
```

Here, `func` takes two arguments. If its first argument equals 1 then it returns 1, otherwise it returns `b`. Now, in a strict language calling this function as

```
func 1 (1 / 0)
```

will result in a divide-by-zero error because `func` will attempt to evaluate both of its arguments before it starts evaluating its body. In a non-strict language, however, the above will never use its second argument, and thus, it will return `1`. Therefore, `func` is said to be non-strict in its second argument.

The overwhelming majority of modern programming languages have strict semantics. Nevertheless, many of them provide options (e.g., Scala's `lazy` modifier [19], Python's `yield` statement [29]) that can be used to accomplish the same effects as in non-strict languages. In general, laziness in an imperative language can sometimes lead to unpredictable results due to side-effects. Examples of call-by-need languages are Miranda [28] and Haskell [17].

There are two practical arguments in favour of call-by-need. The first argument is efficiency; a lazy function computes only what is needed. The second argument is that in a non-strict language, programs can use infinite data structures; and this increases the expressive power and modularity of the language. Below, we give an example that stresses the second argument.

An example In his classic article on *why functional programming matters* [18], Hughes presents programs that solve problems from the areas of numerical analysis and artificial intelligence. One of these programs implements the Newton-Raphson algorithm which computes an approximation of the square root of a real number. Instead of writing the program as a single monolithic function in an imperative language, the author divides it into parts so that each part can be also reused in a different context.

The idea behind this algorithm is that the following sequence converges to the square root of a number a:

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

where x_0 is an initial approximation. So, the first step is to define a function that takes x_n and returns x_{n+1}.

```
next a x = (x + a / x) / 2
```

Having defined `next`, the sequence of approximations can be thought of as a list of successive applications of the function `f = next a`:

$$\texttt{[x}_0\texttt{, f x}_0\texttt{, f (f x}_0\texttt{), ...]}$$

To construct this list, we define a second function that uses `next` and returns the above list.

```
repeat f x = x : (repeat f (f x))
```

Here, `(:)` is the list concatenation operator. The `repeat` function takes a function and another argument, and recursively returns an infinite list whose head element is its second argument, and its tail is the list returned from the call `repeat f (f x)`; or, using a list notation, `[x, f x, f (f x), ...]`. But this is the definition of our approximation list. Notice that `repeat` is a higher-order function that can be used in any context where a list, whose elements are results of successive applications of a function, is needed. Again, in our case, `f = next a`.

Finally, we define another function that is used to extract an element from the list:[3]

```
nth n l = if n == 1 then head l else nth (n - 1) (tail l)
```

The functions `head` and `tail` return the first element, and the list without the first element, respectively. The recursive function above takes a number n and a list, and returns the n^{th} element of the list.[4]

Putting all the pieces together, calling `nth` as follows:

```
nth 10 (repeat (next 2) 1)
```

returns the tenth element of a list that consists of successive approximations to the square root of 2, beginning from the initial approximation 1. The key point here is that if our language were strict, the above expression would not terminate. But in a lazy language, the function `nth` evaluates exactly the part of the list that it needs. To understand how laziness is crucial in this program, we unfold the recursion tree of the execution. We start by rewriting the first call:

```
nth 10 (repeat (next 2) 1) ⇒
if 10 == 1 then head l else nth (10 - 1) (tail l)
```

where `l = (repeat (next 2) 1)`. Of course, the condition fails and the execution continues with the next call:

```
nth (10 - 1) (tail l)
```

Note that so far, no argument of `nth` has been evaluated. Actually, the first argument is evaluated later, when `nth` checks if the condition holds, i.e., if `(10 - 1) == 1`. Again, the condition fails and `nth` is called recursively as

```
nth (9 - 1) (tail (tail l))
```

[3]Choosing `nth` instead of a function that returns the element from the list where the difference between two consecutive approximations is less than a predefined number, to demonstrate laziness simplifies our analysis below.

[4]For simplicity reasons, we do not handle exceptional cases such as when the index is greater than the length of the list.

Another important thing here is that since the first argument of `nth` was evaluated in the previous call, it does not have to be re-evaluated and thereby, 9 is used instead of (10 - 1). As the evaluation continues and the recursive calls are expanded, the leaf of the tree is reached:

```
nth (2 - 1) (tail ( ... (tail l)))
```

This time, the condition succeeds and the call returns

```
head ((tail ( ... (tail l)))
```

But again, both `head` and `tail` are lazy. Thus, the whole expression above is wrapped up in a *thunk* [13] and is propagated back to the recursion root. Finally, it is returned as the result of the program. Note that the program itself does not evaluate this expression when it terminates; it returns it as it is: an unevaluated expression. Only if another program required its result, would the expression be evaluated. For example, applying a function, such as `print`, would force its evaluation. In this case, `head` would be applied to the list

$$\texttt{i}^* \texttt{ : repeat (next 2) i}^*$$

which is the result of the successive calls of `tail` to `l`. Here $\texttt{i}^*$, which is the tenth approximation of the square root of 2 (or the tenth element of the infinite list), would be printed.

Clearly, the elegance of the lazy implementation of the Newton-Raphson method above stems from the way it is modularized. In particular, data is separated from control, rendering the two concepts orthogonal to each other. More precisely, the core function of the algorithm is plugged into another function which iteratively composes the core function with itself, and returns an infinite list. Of course, we could just as well have used another core function. A third function — the consumer — takes this list and evaluates only the part that it needs.

Evaluation Contexts

At the beginning of this section we said that the evaluation order of a programming language is expressed as a set of rules. Next, we described call-by-value, call-by-name, and call-by-need in a somewhat informal manner. In the following paragraphs we study evaluation strategies in a more formal way.

The rules that define the reduction order of the call-by-value λ-calculus are the following:

$$(1)\frac{t_1 \rightarrow_\beta t_1'}{t_1\ t_2 \rightarrow_\beta t_1'\ t_2} \qquad (2)\frac{t_1 \rightarrow_\beta t_1'}{v\ t_1 \rightarrow_\beta v\ t_1'}$$

where $\rightarrow_\beta$ is the standard reduction relation (β-axiom) of the calculus and t_i are terms. Rule (1) states that in an application expression, the left-hand side term can always be reduced. Rule (2) states that in an application expression, the right-hand side term can be reduced only if the left-hand side term is a value.

An alternative way of expressing evaluation order rules is with *evaluation contexts*. An evaluation context, denoted by $E[\ ]$, is a term (or expression) with a *hole* in it. If $E_1[\ ]$ is an evaluation context and t is a term, then $E_1[t]$ is the expression that derives from the evaluation context whose hole is replaced with t. For example, let $E_1 = [\ ]+3$ and $t = 1+2$. Then, $E_1[t] = (1+2)+3$. The important thing here is that we can expressly specify which expression must be evaluated next by decomposing a term into an evaluation context and the next redex. Once this redex is reduced, its result is placed in the hole of the context. This can be made explicit with the following rule:

$$\frac{t \rightarrow_\beta t'}{E[t] \rightarrow_\beta E[t']}$$

This *general context rule* states that if a term can be decomposed into an evaluation context $E[\ ]$ and a redex t, and t can be β-reduced to t', then we can proceed with the reduction and place the result t' into $E[\ ]$. Having this rule, we can define the set of evaluation contexts for the call-by-value λ-calculus as:

$$E ::= [\ ] \mid E\ t \mid v\ E$$

As an example that illustrates the concepts described above, consider the term

$$(\lambda x.x)\ (1+2)$$

We can express this term as $E_1[t]$, where $E_1 = (\lambda x.x)\ [\ ]$ and $t = (1+2)$. Obviously, E_1 has the form of $v\ E$. Thus, using the general context rule, it becomes clear that t has to be evaluated before the function is applied to it. Equivalently, we could use rule (2) to obtain the same result.

Evaluation contexts are useful, not only for defining evaluation order rules, but also for representing continuations, as we shall see in the next chapter. In general, the concept of evaluation context will be coming up very often throughout this thesis. Most importantly, in the last chapter evaluation contexts are used to simulate store-based effects in an abstract machine.

2.3 Algebraic Data Types

Algebraic data types are a powerful data abstraction mechanism of programming languages that allows us to define new types. An algebraic data type definition abides by the following syntax rule:

$$
\begin{array}{lll}
\textbf{data}\ \mathtt{T} & = & \mathtt{C}_1\ \mathtt{T}_{1,1}\ \ldots\ \mathtt{T}_{1,m_1} \\
 & | & \ldots \\
 & | & \mathtt{C}_n\ \mathtt{T}_{n,1}\ \ldots\ \mathtt{T}_{n,m_n}
\end{array}
$$

where T is the new type, also called *type constructor*, $T_{i,j}$'s are types and C_i's are *data constructors* with arity m_i. A value is of type `T` if it has been created using one of `T`'s data constructors.

Another way to look at algebraic data types is as a *sum-of-products* [21]: a data constructor with its fields can be thought of as a product type (or *tuple*) while the set of the alternatives of constructors can be thought of as a sum type (or *disjoint union*), in the sense that a value of type `T` can be constructed by (exactly) one of the $\mathtt{C}_i$'s.

A data constructor can have no arguments, i.e., its arity can be 0. In this case, it is a *nullary* constructor. For example, in some functional languages the type `Bool` is defined as:

```
data Bool = True | False
```

Here, `Bool` is a user-defined type with two nullary data constructors: `True` and `False`. Types that contain only nullary constructors are called *enumeration types*.

Algebraic data types can also be recursive, that is, a type can refer to itself on the right-hand side of the definition. As an example, below we define a recursive `Tree` type that represents binary trees for integers:

```
data Tree = Node Integer Tree Tree
          | Leaf Integer
```

Assuming `Integer` is the built-in type for integers, a value of type `Tree` can be either a `Node` that has a value of type `Integer` and two values of type `Tree`, or a `Leaf` that has just a value of type `Integer`. Put differently, the `Tree` type is the sum of two products: the product (`Integer` × `Tree` × `Tree`) and the trivial product that has a single field of type `Integer`. The following is an example of a value of type `Tree`:

```
Node 1 (Node 2 (Leaf 3) (Leaf 4)) (Leaf 5)
```

The node with the value 1 is the root of the tree. This node has two branches: one is another node and the other is a leaf, with the value 5.

Along similar lines, we can define other recursive types such as lists. Most importantly, with algebraic data types, lists and trees — which are essential to functional programming — need not be built into the language; rather, we can define them ourselves in a concise, clean, and intuitive way. Moreover, we can use standard mathematical methods, such as induction, to formally reason about them.

2.4 Pattern Matching

In the previous section we saw how one can define a new type using algebraic data types. Yet, we did not see how to deal with these user-defined types, and having such a powerful feature requires the proper tool to make the most of it. Pattern matching is the standard *recognition* and *deconstruction* tool used in conjunction with algebraic data types. More precisely, pattern matching is the means by which we can:

- Inspect the structure of values and determine which data constructor has been used for their construction.

- Extract the components from a value and bind them to variables.

In general, a pattern matching expression has the following syntax:

```
case expr of
    pat₁ -> expr₁
      ⋮
    patₙ -> exprₙ
```

where `expr` is an expression, and each line of the form `pat`$_i$ `->` `expr`$_i$ is called a *branch*, and consists of a *pattern* and an *expression*. A pattern is a data constructor whose fields are variables which we will call *pattern variables*.[5]

Informally, pattern matching evaluates the expression `expr` and checks the result against each of the patterns `pat`$_i$ starting from the topmost branch (we assume that evaluating `expr` results in a constructor value). If the pattern's constructor is the same as that of the expression, then we say that the pattern match succeeds. When the first successful match occurs, the expression of the right-hand side of the branch, whose pattern match succeeded, is evaluated. The result of this evaluation is the final result of the whole pattern matching expression. If no pattern match succeeds, then the program evaluation stops with an error.

As we said above, pattern matching is the standard deconstruction tool. When a match succeeds, the components of the `case` argument are bound to the variables of the successful pattern and can thereby be used when the expression part of the branch is evaluated.

As an example, consider the following program:

```
data NewType = Cons1 Integer
             | Cons2 Integer


case Cons2 3 of
    Cons1 x -> x + 1
    Cons2 x -> x * 2
```

Here, we first define a new type called `NewType` that has two data constructors: `Cons1` and `Cons2`, each of which has one integer field. Next, we pass a value of this type, which has been constructed with the `Cons2` data constructor, to a pattern matching expression

[5]In general, a pattern can be a constant (e.g., a number or a character), a variable, or a constructor whose elements are patterns. In this work, however, we consider only the case in which a pattern is a constructor whose elements are variables.

with two branches. The first attempt to match the `case` argument against `Cons1` fails but the second attempt to match it against `Cons2` succeeds. Then, the component of the value constructed with `Cons2`, which is the integer value 3, is bound to the variable of the successful pattern, `x`, and the expression `x * 2` is evaluated; the final result of the whole pattern matching expression is 6.

Pattern matching in a strict language behaves differently from a non-strict language. We call the former *strict pattern matching* and the latter *non-strict pattern matching.* Strict pattern matching semantics differs from non-strict in that, in the first case, the expression passed to `case` is fully evaluated before pattern matching proceeds. That is, if the expression is a constructor, all of its components are evaluated. On the other hand, in non-strict pattern matching, a constructor's argument is only evaluated if its value is needed inside a branch expression. Therefore, when a match succeeds pattern variables are bound to unevaluated expressions. We call a constructor whose arguments are not evaluated *lazy.* In general, a constructor whose arguments may or may not be evaluated is said to be in *weak head normal form* (WHNF) [20]. In this work, we focus on non-strict semantics and its extension to pattern matching.

To explain non-strict semantics in the context of pattern matching, assume that we have a recursive binding mechanism, such as Haskell's `let`, which we use in conjunction with the type definition of `Tree` we introduced in the previous section, to construct a value:

```
let inftree =
    Node 1 (Node 2 (Leaf 3) (Leaf 4)) inftree
```

Note that this is a recursive definition and results in an infinite data structure. Here, we define a tree whose root has two children. The left child has two children which are both leaves. The right child recursively refers to the tree itself. In other words, the tree's rightmost branch is infinite.

In a strict language, this definition would lead to non-termination. But a lazy language will evaluate only the parts that are needed.

To illustrate the difference, we define a function that takes a tree argument and checks its data constructor; if it is a node, it binds the pattern variables to the node's components. That is, `i` is bound to the number, and `l` and `r` are bound to the left and right sub-tree, respectively. Then, the function recursively calls itself passing it the left sub-tree. If `t` is a leaf, it just returns its number. This function essentially returns the number of the leftmost leaf; and in doing so, it evaluates only parts of the sub-trees from which it passes.

```
leftmost t = case t of
                Node i l r -> leftmost l
                Leaf n     -> n
```

Calling the above function and passing `inftree` to it as below:

```
leftmost inftree
```

yields 3. The first time `leftmost` is called, `inftree`'s outermost constructor is checked against the pattern of the first branch but it does not evaluate its components. The match succeeds and the pattern variables are bound to `inftree`'s components: `i` is bound to 1, `l` is bound to `(Node 2 (Leaf 3) (Leaf 4))`, and `r` is bound to `inftree`. But the second `leftmost` call uses only the `l` variable, so `i` and `r` are not used at all and are thus discarded without ever being evaluated. The computation proceeds with the next call, in which it finds yet another node, calls `leftmost` again, this time matching the leaf pattern, and finally returns 3.

Lazy pattern matching inherits all the advantages of lazy evaluation. In the example of Section 2.2 we demonstrated the readability and conciseness of lazily evaluated programs with the implementation of the Newton-Raphson algorithm, where we used two functions — `head` and `tail` — to manipulate an infinite list. However, we took for granted that these functions somehow exist and we did not disclose any detail about their implementation. We just relied on their laziness as we did for the functions we defined ourselves. In fact, lazy pattern matching, in tandem with algebraic data types, makes the definition of functions, such as `head` and `tail`, simple, clean, and natural.

Pattern matching is, in general, regarded as an inseparable feature of functional programming languages. Lazy pattern matching, in particular, combines the usefulness of pattern matching with the elegance of non-strict semantics. In this work, we study the relationship of lazy pattern matching with the subject of the next chapter: continuations.

Chapter 3

Continuations

The subject of this chapter is continuations. In Section 3.1 we give an insight into the fundamental concepts, and in Section 3.2 we explore continuation-passing style: a style of programming that makes the control flow of a program explicit. Sections 3.3 and 3.4 introduce control operators which are mechanisms that allow us to manipulate continuations in programs written in direct style. Finally, in Section 3.5 we present delimited control operators which are more general and expressive than undelimited control operators, and study some well-known proposals.

3.1 Fundamentals

A *continuation* is the rest of the computation at any given moment. Continuations exist naturally in all programming languages. When evaluating an expression that is part of a program, the continuation at this particular point is the rest of the program waiting for the result of this expression. A continuation can be thought of as a program with a hole in it which is filled with the value of the currently evaluated expression once the latter is completed. We denote this hole with a $\Box$. For example, in the program `(1 + 2) + 3`, the expression `1 + 2` has to be evaluated first and its value is passed to another expression, whose outcome is the value plus 3. In other words, the continuation of the expression `1 + 2` in this program, is `□ + 3`. Clearly, a continuation can be regarded as a one-argument function. Therefore, in our simple example, the function $\lambda x.x + 3$ functionally represents the continuation □ `+ 3` of the expression `2 + 1`. Invoking this continuation and passing it a value is equivalent to applying the function that represents the continuation, to the passed value.

3.2 Continuation-Passing Style

In the traditional way of structuring programs, when a function is called, it performs its computation and then returns its result to the point it was called. This style of programming is referred to as *direct*. *Continuation-passing style* (CPS) [27], on the other hand, is a programming style in which all functions are defined so that they take an additional parameter — a continuation. This continuation is a function which is applied to the computed value, and represents what has to happen from that point until the end of the execution of the program.

In CPS, functions do not return. Instead, they hand their result to their continuation. A program written in CPS is a computation that consists of a chain of function calls. The last action of each of these functions, after computing their value, is apply the function that stands for their continuation to their result. The restriction that every function performs its computation and then passes its result on to the next function makes control and data flow of a program explicit and saves us from keeping track of things such where a function should return.

Any program written in direct style can be transformed into CPS automatically. As an example, let us define two functions (in Haskell): `append` and `sumlist`. The first function appends a list to another, while the second returns the sum of the elements of a list. We also define a third function called `appendsum` that uses the composition of `append` and `sumlist`, and returns the sum of the elements of two concatenated lists. In direct style, these functions can be written as follows:

```
append :: [Int] -> [Int] -> [Int]
append l1 l2 =
  case l1 of
    []   -> l2
    x:xs -> x : append xs l2
```

```
sumlist::[Int] -> Integer
sumlist l =
  case l of
    []    -> 0
    x:xs -> x + sumlist xs


appendsum::[Int] -> [Int] -> Int
appendsum l1 l2 =
  sumlist (append1 l1 l2)
```

Listing 3.1: Direct style.

Apparently, the `append` and `sumlist` functions are recursive. The function `appendsum` first calls `append`, passing it its arguments, and then applies `sumlist` to the value returned from the first call. Finally, the value returned from `appendsum` is the value returned from `sumlist`.

To transform these functions into CPS, we rewrite them so that they take an additional parameter, the continuation, which they use to apply to the value they compute. This parameter is a function that represents the rest of the computation.

```
append :: [Int] -> [Int] -> ([Int] -> t) -> t
append l1 l2 k =
  case l1 of
    []    -> k l2
    x:xs -> append xs l2 (\a -> k (x : a))


sumlist :: [Int] -> (Int -> t) -> t
sumlist l k =
  case l of
    []    -> k 0
    x:xs -> sumlist xs (\a -> k (x + a))


appendsum :: [Int] -> [Int] -> (Int -> t) -> t
appendsum l1 l2 k =
  append l1 l2 (\a -> sumlist a k)
```

Listing 3.2: Continuation-passing style.

In all three functions the parameter `k` is the continuation. Note that in both `append` and `sumlist`, after turning them into CPS, recursive calls became *tail calls.* In addition, intermediate results that were implicit in direct-style, in CPS, their computation becomes explicit. These distinctive features of CPS make it a suitable form on which several optimizations, such as *tail-recursion elimination*, can be applied. An extensive and thorough study of CPS as an intermediate representation and its benefits to optimizing compilers can be found in Appel's book [2].

CPS has found numerous — theoretical and practical — applications in the field of programming languages. A notable example of the first kind is denotational semantics. In their seminal work [26], Strachey and Wadsworth used CPS as the foundation to build a mathematical model which describes the behaviour of programs that contain non-local jumps, i.e., a mechanism that allows the programmer to alter the control flow of a program by defining functions that can be used to exit a function or escape while evaluating an expression. In their model, they introduce two semantic functions, $\mathcal{P}$ and $\mathcal{E}$ which are based on a set of equations. These functions translate terms of the language into their meaning. The language comprises both expressions and statements (commands). $\mathcal{P}$ gives a semantic interpretation to the former while $\mathcal{E}$ is for the latter. In both cases, however, the meaning of a program is taken to be a state transformation, i.e., a function from a program state to another. Moreover, besides the standard arguments, an environment and a store, $\mathcal{P}$ and $\mathcal{E}$ take a continuation. Given a term of the language, this continuation represents the state transformation that corresponds to the rest of a program after the term has been evaluated. If the term contains a transfer of control, along with a label denoting the destination, then the passed continuation is replaced with the continuation that corresponds to this label. This type of denotational semantics, which is also known as *continuation semantics*, has been one of the standard models used in the design of programming languages. Moreover, despite its simplicity, the model can be easily extended to describe a wide set of terms and language features.

Another remarkable work that introduced continuations in the definition of functional programming languages is that of Reynolds [23]. In this highly-influential paper, a *meta-circular* interpreter, which is written in a language called the *defining* (or *target*) language, is used to define another, called the *defined* (or *source*) language. Here, meta-circularity means that the features of the source language are defined in terms of the corresponding features of the target language. This implies that if the order of evaluation of the target language changes, e.g., from call-by-name to call-by-value, so does the source language's, which in turn alters the meaning of some (but not all) programs.

Initially, the interpreter is written in direct style, but this approach suffers from the shortcoming that the programs it evaluates are dependent on its own evaluation order. That is, functions in the source language evaluate their arguments in the same order as the target. To overcome this restriction, the interpreter is transformed into CPS.

In his paper, Reynolds uses the CPS transformation so that the source language evaluates functions' arguments in a call-by-value order, regardless of the evaluation order of the language in which the interpreter itself is written. The same method, however, can also be used to define a call-by-name language. The transformation is motivated by the observation that if the order of evaluation is made an explicit part of a program's structure, it becomes independent of the evaluation order of the program's interpreter.

The interpreter has the form of a double-argument evaluation function, called `eval`, whose execution depends on the type of its first argument: the expression that is evaluated. The second argument of `eval` is the *environment*. The environment is a data structure in which variable bindings to values are stored. The `eval` function is essentially an abstract machine that gives the operational semantics of the defined language.

Consider the case in which the `eval` function evaluates a function application expression. That is, the part of the interpreter that applies a function of the source language to an argument:

$$\texttt{eval}(r_0\ r_1, e) \Rightarrow (\texttt{eval}(r_0, e))\ (\texttt{eval}(r_1, e))$$

Here, the expression that is evaluated is $r_0\ r_1$, where r_0 and r_1 are a sub-expression that is expected to yield a function, and the argument, respectively, and e is the environment. Obviously, the order of evaluation of the target language affects the order of evaluation of the source language. For example, if the target language follows a call-by-value strategy, then r_0 is evaluated first, and if the result is a function, then r_1 is evaluated. Finally, the body of the function returned from r_0 is applied to the value returned from r_1. Consequently, call-by-value will also be the evaluation order of the source language. If, on the other hand, a call-by-name strategy is used by the target language, r_0 is again evaluated first but, as opposed to call-by-value, r_1 will not be evaluated unless its value is required during the evaluation of the function's body returned from r_0.

Fortunately, we can loosen this tight coupling, by using the CPS transformation to embed the evaluation order within the program. For example, using a call-by-value strategy for the source language means that the interpreter has to carry out the following steps in this specific order:

1. Evaluate r_0.
2. Evaluate r_1.
3. Apply the value of r_0 to the value of r_1.

Thus, `eval` has to be rewritten so that the above execution order is encoded into its structure: as soon as `eval` identifies that the expression is a function application, it evaluates r_0 and passes the result to a function that evaluates r_1, which in turn passes the result to another function that applies r_0 to r_1. Encoding this sequence of steps into the `eval`'s code, results in the following form:

$$\begin{aligned}&\texttt{eval}(r_0\ r_1, e, c) \Rightarrow\\ &\quad \texttt{eval}(r_0, e, \lambda f.\texttt{eval}(r_1, e, \lambda v.\texttt{eval}(\mathit{body}(f), \mathit{ext}(e, f, v), c)))\end{aligned}$$

Here, if f is a function, $\mathit{body}(f)$ returns its body. Moreover, $\mathit{ext}(e, f, v)$ extends the environment e by adding a binding of the formal parameter of f with the value v.

When an application expression $r_0\ r_1$ is met, `eval` evaluates r_0 and passes the result to $\lambda f.\texttt{eval}(r_1, e, \lambda v.\texttt{eval}(\mathit{body}(f), \mathit{ext}(e, f, v), c))$. In this function, `eval` is called once again, this time with r_1 as its first argument and $\lambda v.\texttt{eval}(\mathit{body}(f), \mathit{ext}(e, f, v), c)$ as the continuation. Finally, the last call to `eval` performs the application and passes the result to c. Evidently, in this sequence no call to `eval` ever returns to the point it is called.

The additional parameter `eval` takes is the continuation. Notice that the parameter c of the `eval` function is also used in the innermost call. This is the continuation passed to `eval` when it is invoked and can be any function that takes a value and returns a value. Typically, the function $\lambda x.x$, i.e., the identity function, is used to bootstrap the evaluation.

A similar approach as the one presented above but with a different continuation function could be used to define call-by-name. For example, we could have defined the order of call-by-name and encode it directly in the `eval` function just as we did with call-by-value.

The method used by Reynolds to eliminate the evaluation order dependence was later formalized in the context of the λ-calculus by Plotkin, and it is referred to in the literature as the *Indifference Theorem* [22, 13].

3.3 Control Operators

Thus far, we have only seen continuations that are represented by ordinary functions and thereby, calling or constructing them does not require any special mechanism from the language, apart from higher-order support. Therefore, using continuations in a program implies that this program is written in CPS. Continuations in a CPS-written program are often referred to as *explicit*. In this section, we explore operators that are used to manipulate continuations *implicitly*. More precisely, in the paragraphs that follow we study mechanisms that allow us to interact with continuations in a program that is written in direct-style.

In Section 3.2 we mentioned that when an expression is evaluated, its value is passed to the remainder of the program — its continuation. In a sense, the continuation is a functional representation of the context (or the *control stack*) surrounding this expression, and control operators allow us to gain access to this context. The type of access, however, varies greatly. Some control operators just *discard* (or *abort*) this context. Others, capture it and hand it to the programmer as an abstraction, in the form of a function; a process which is known as *reification*. In the following section we describe some of the most widely-studied control operators. Note that there are numerous variants of these operators in the literature and this thesis does not provide a complete list.

A basic control operator which is present in many programming languages is `abort`. This function takes one argument, and when it is called, it evaluates its argument and then, it aborts (or deletes) the current continuation. The final result of a call to `abort` is the result of the evaluation of its argument. The behaviour of `abort` is defined by the following reduction rule:

$$E[\texttt{abort}\ e] \Rightarrow e$$

where E is an evaluation context and e is an expression. From the right-hand part of this rule, we see that the evaluation context, in which `abort` is invoked, is removed completely and what remains is the evaluation of e. In other words, calling `abort` inside an expression passing it another expression discards the first expression altogether and returns the one that is passed to it as its argument.

3.4 First-Class Continuations

The `abort` operator, described above, removes all the information stored in the control stack at a given moment and thereby, discards the entire evaluation context. We would like, however, to save this information so we can use it later. *Higher-order* control operators store the context and hand it to the user as a functional abstraction. With these operators, we can control *first-class* continuations, that is, continuations that can be passed to and returned from functions, be saved, invoked and in general, handled just like any other first-class object of the language.

A widely-studied control operator of this kind, which exists in most of the functional programming languages, is `call-with-current-continuation` (abbreviated `call/cc`). The `call/cc` function — which is similar to `escape` (see [23]) — first appeared in Scheme [7]. This operator captures the current continuation and binds it to a variable, enabling the programmer to call it like any other function. (Although, as we will see below, a continuation captured by `call/cc` lacks a fundamental property of functions: *composability.*) Calling a continuation captured by `call/cc` results in the replacement of the current with the stored context. Thus, a continuation can be regarded as a jump to the point where it was captured.

The `call/cc` function takes one argument: a function. When it is called, it captures the current continuation and binds it to the formal parameter of its argument. When this parameter is invoked with an argument passed to it, it discards the current continuation, and restores the captured continuation, applied to its argument. Below is the reduction rule for `call/cc`:

$$E[\texttt{call/cc}\ (\lambda k.M)] \Rightarrow E[(\lambda k.M)\ (\lambda v.\texttt{abort}\ E[v])]$$

where, again, E is an evaluation context. To give an explanation of this rule, `call/cc`'s argument is the function $\lambda k.M$. Here, M denotes the body of the function. When `call/cc` is invoked, it binds the function $\lambda v.\texttt{abort}\ E[v]$ to k and it goes on to evaluate M. If, during M's evaluation, k is applied to a value, e.g., $k\ v$, then `abort` discards the current continuation, and the captured continuation is restored with v in its hole, i.e., $E[v]$. An important thing here, however, is that since k's invocation removes the current continuation, k cannot be composed with itself or any other function, simply because it does not return where it was called; rather, it returns to the point where `call/cc` was called. Therefore, k is not an ordinary function, and continuations captured by `call/cc` are said to be *non-composable* or *abortive* as opposed to *composable* or *delimited* continuations that we will look at shortly.

The simple Scheme program below exemplifies `call/cc`'s behaviour:[1]

```
(+ 1 (call/cc
  (lambda (k)
    (+ 2 (k 3)))))
```

Listing 3.3: A simple `call/cc` example.

In this example, `call/cc` is called with the function `(lambda (k) (+ 2 (k 3)))` as its argument. First, `call/cc` captures the current continuation, which, at the moment of its call is `(+ 1 □)`, and binds it to `k`. Then, it proceeds with the evaluation of the lambda's body, where `k` is first applied to `3`. Now, the result of this expression would normally be passed to its continuation, i.e., `(+ 2 □)`, which is the current continuation when `k 3` is evaluated. But this is not the case! When `k` is applied to 3, the current continuation is discarded, and the captured continuation is reinstated, with 3 passed to it. Thus, the captured continuation `(+ 1 □)` is instead applied to `3`, and the final result of the program is `4`.

[1]In this chapter we use Scheme to write the examples because of its simple and clean syntax, as well as its extended support of control operators.

As we said at the beginning of this section, a continuation can be stored and used outside `call/cc`. To illustrate this, let us consider the following example:

```
(define cont #f) ;; Dummy initialization of a global variable

(+ 1 (call/cc
  (lambda (k)
    (set! cont k)
    (+ 2 (k 3)))))

(+ 1000 (cont 1))
```

Listing 3.4: Saving a continuation.

In this slightly different program, we first define a global variable and then run `call/cc` again, but this time, the function passed to it stores the continuation, bound by `call/cc`, to the global variable before doing the same computation as in the previous example. Again, this expression yields 4. Next, we add the result of the `(cont 1)` to 1000. But once we call `cont`, the current continuation (i.e., `(+ 1000 □)`) is discarded and the captured continuation is restored. Thus, the final result is 2.

Going back to the previous section for a while, where we stated that continuation semantics is the standard model for reasoning about control operators, we can give a semantic equation for `call/cc` as follows:

$$\mathcal{E}[\![\texttt{call/cc}\ \ (\lambda x.e)]\!]\rho\kappa = \mathcal{E}[\![e]\!]\rho\{x \mapsto \lambda v.\lambda\kappa'.\kappa\ v\}\kappa$$

In this equation, ρ is the environment and κ is the continuation passed to the semantic function (we deliberately omit the store argument since we assume that the language we denote is functional). What this equation says is that the meaning of a `call/cc` invocation, with a function as its argument, is the meaning of the function's body, where ρ is extended with a mapping from x to another function. This last function takes two arguments: a value v and a continuation κ'. When this function is called, it passes v to the continuation κ, which was initially passed to $\mathcal{E}$, but discards the current continuation (bound to κ').

Coroutines and Continuations

The `call/cc` operator has been used to model jumps in a variety of applications. Among them is an implementation of coroutines proposed by Haynes, Friedman, and Wand [15]. In this work, a coroutine framework is built using first-class continuations. Avoiding the introduction of other complicated mechanisms into the language and relying on the intuitive notion of continuations helps to greatly reduce the complexity of the framework.

Coroutines are *collaborative* program components that keep their own local state as well as share data between each other. A coroutine is said to be *non-preemptive*, if it voluntarily suspends its own execution and passes control to another coroutine. Each coroutine must store its current state of computation so that when it is resumed, it can continue from the point it was suspended.

The continuation-based framework models coroutines with functions. Each function, which has a local state, implements its private `resume` function. The `resume` function takes two arguments: the resuming coroutine and a value passed to it. When called, `resume` records the current state of its coroutine. This current state is the coroutine's continuation which is captured by `call/cc` and is stored in a local variable. After having captured the current state, `resume` suspends the execution of its coroutine and invokes the resuming coroutine.

Logic and Continuations

Another notable work that uses first-class continuations and `call/cc` in particular is Haynes's backtracking method [14]. This method combines two programming paradigms, namely logic and functional programming, to solve complex problems from artificial intelligence.

Often enough, in applications of logic programming the programmer needs to impose a search strategy, which may depend on various runtime factors, rather than rely on a language's default strategy. One way to define such a strategy is with the manipulation of the control context. This, however, requires that the programmer has access to an appropriate control abstraction mechanism, such as first-class continuations, that allows the definition of a non-standard control behaviour. In general, logic programming languages do not provide such a mechanism. Therefore, the programmer has to resort to other programming paradigms such as functional programming.

In general, an *embedding* is an implementation which translates a language, called the *embedded* language, into another language, called the *embedding* language, so that the former can take advantage of the facilities of the latter. In his article, Haynes proposes an embedding of a logic programming language (e.g., Prolog) into a functional programming language (e.g., Scheme). Thus, a program written in the embedded language can exploit the control abstraction mechanism of continuations of the embedding language — or any other feature for that matter — whenever that is required.

In particular, the problem that the embedding of Haynes's article addresses is called *non-blind backtracking* and it is essentially the ability of a system to discard, yet remember the state of a context that took place in the past. In other words, non-blind backtracking means that when a computation backtracks from a given point to another that was previously visited, it has to record the state from the discarded point in case it needs to transfer the control back to it. As an example, consider a program that starts in a context A and proceeds to another context B. There, an action takes place that alters the program's state, and the program moves to another context, C. But while being in C, the program realizes that the option made in B was not the optimal and decides to backtrack to B, undo the action and choose a different context, D. However, this path turns out to be even worse; so the program moves back to C. But now the program's state needs to be restored to that of its previous visit to C.

3.5 Delimited Control Operators

In the previous section, we explained what a control operator is and how it allows a programmer to capture and reinstate the remaining part of a computation. Moreover, we described how `call/cc` works and gave an overview of some of the most notable examples in which it is used to solve real-life problems.

The `call/cc` control operator, however, is restricted to capture the *entire* rest of the computation. In this section, we concern ourselves with more powerful operators; that is, operators that let us capture part of the rest of a computation rather than the whole of it.

As is revealed by their name, *delimited control operators* are used to delimit a captured continuation. In other words, before turning the control context into an abstraction, these operators allow us to mark a point, up to which the context extends. In this section, we focus on this kind of operators, we examine their operational characteristics and finally, we present some of the most widely-studied proposals.

In many languages, the `abort` operator is accompanied with an additional mechanism that allows the programmer to delimit the context that is aborted. Put simply, calling `abort` in a delimited context aborts part of the continuation instead of discarding the entire control stack. This delimited `abort` is the most basic form of delimited control operators and constitutes the basis for an exception handling system such as, for example, OCaml's `try/with` construct along with the `raise` function.

The delimited `abort` operator, like its undelimited counterpart, does not offer any significant power to the language. As we said above, in order to take full advantage of continuations we would like them to have first-class status. In the paragraphs that follow we present delimited control operators that, unlike delimited `abort`, give us access to first-class continuations.

Usually, delimited control operators come in pairs; one of them is the *reifier*, while the other is the *delimiter*. The former captures and binds the continuation to a variable and the latter specifies the point up to which the captured continuation extends. One such pair is `shift/reset` by Danvy and Filinski [8].

The `shift` operator takes two arguments. When called, it captures the current continuation, which is delimited by the innermost enclosing `reset`, binds it to its first argument and evaluates its second argument. Obviously, the ability to capture a segment of the continuation makes it more general than `call/cc`. There is, however, another significant difference regarding the functional abstraction that represents the captured continuation. Rather than being non-composable, as is the case with `call/cc`, the continuation captured by `shift` can be composed just like an ordinary function. In other words, the continuation returns to the point where it was called; not where it was captured. Thus, continuations captured with `shift/reset` are called *delimited* or *composable* continuations (or simply *subcontinuations*).

The following two rules define the operational behaviour of `shift/reset`:

$$\texttt{reset}\ v \Rightarrow v$$

$$\texttt{reset}\ (E[\texttt{shift}\ k\ e]) \Rightarrow \texttt{reset}\ ((\lambda k.e)\ (\lambda x.\texttt{reset}\ E[x]))$$

Here E is an evaluation context, e is an expression and v is a value. According to the first rule, `reset` applied to a value returns the value. The second rule says that `shift` refies the continuation delimited by the innermost `reset` and binds it to k. Then, it proceeds with e's evaluation. If, during the evaluation of e, k is applied to a value, the result of the refied continuation applied to this value is returned. If, on the other hand, k is not used at all, then the captured continuation is discarded. In both cases, `reset`'s result is the value returned from `shift`. Consider the following example, taken from [24], that illustrates the above rules:

```
(cons 1
  (reset
    (cons 2
      (shift k (cons 3 (k (k (cons 4 '()))))))))
```

Listing 3.5: `shift/reset` example.

In this program, the continuation captured by `shift`, which is (`cons 2 □`), is bound to `k` and it is first applied to the result of `(cons 4 '())`, (or `'(4)`). Then, `k` is once again invoked, this time composed with itself. That is, the continuation (`cons 2 □`) is applied to `'(2 4)`. Next, `3` is appended to the head of the list of the last application. Finally, the resulting list, which is `'(3 2 2 4)`, is passed to the current continuation, i.e., `(cons 1 □)`, and the final result of the program is `'(1 3 2 2 4)`.

In Section 3.2 we said that a program written in direct style can be automatically transformed into CPS. Of course, the same holds for the λ-calculus. In particular, the translation function for the call-by-value λ-calculus is given in Figure 3.1. ($t_1\{x \mapsto t_2\}$ denotes the substitution of t_2 for x in t_1.) The `shift/reset` operators were originally defined as an extension of this transformation with the aim of harnessing the *unused* expressiveness of CPS [8, 9]. Obviously, these operators are not expressed in continuation-passing style, but rather in *continuation-composing style*. This means that in the definitions

$$\overline{x} = \lambda\kappa.\kappa\ x$$

$$\overline{\lambda x.e} = \lambda\kappa.\kappa\ (\lambda x\kappa'.\overline{e}\ \kappa')$$

$$\overline{e_1\ e_2} = \lambda\kappa.\overline{e_1}\ (\lambda f.\overline{e_2}\ (\lambda v.f\ \kappa\ v))$$

$$\overline{\texttt{reset}\ e} = \lambda\kappa.\kappa\ (\overline{e}\ (\lambda x.x))$$

$$\overline{\texttt{shift}\ k\ e} = \lambda\kappa.\overline{e}\{k \mapsto (\lambda a\kappa'.\kappa'\ (\kappa\ a)\}(\lambda x.x)$$

Figure 3.1: The call-by-value CPS translation of the λ-calculus with `shift`/`reset`.

of `shift`/`reset`, the continuation calls are not tail-calls. Thereby, the evaluation order-independence of CPS that we described in Section 3.2 is lost. To overcome this problem, Danvy and Filinski transformed once again the definitions of Figure 3.1 — a method called *extended continuation-passing style* (ECPS) — which turned all continuation calls into tail-calls. Apart from the usual continuation, the second translation introduced another continuation, called meta-continuation, which in the case of `shift`/`reset` represents the context that is outside of the captured continuation. That is, it is that part of the context that is *not* captured by a call to `shift`. The method of ECPS can be also generalized to continuation semantics so that a formal description based on continuations is given to delimited control operators.

The `shift`/`reset` operators are said to be *static* in that the extent of the captured continuation is determined statically, as opposed to the `control`/`prompt` operators which were introduced by Felleisen [11] and are said to be *dynamic*. In Felleisen's proposal, the `prompt` operator is the delimiter while the `control` operator is the reifier. These operators resemble `shift`/`reset`. However, as can be seen from the reduction rules below, there is a subtle difference between `shift`/`reset` and `control`/`prompt`:

$$\texttt{prompt}\ v \Rightarrow v$$

$$\texttt{prompt}\ (E[\texttt{control}\ k\ e]) \Rightarrow \texttt{prompt}\ ((\lambda k.e)\ (\lambda x.E[x]))$$

Apparently, these rules are similar to `shift`/`reset`'s. They only differ in that `shift` encloses the captured continuation in a `reset` whereas `control` does not. This restricts the continuation in the former case from accessing its outer context. Put differently, when a continuation is captured inside another continuation that has been captured by `shift`, the enclosed continuation's extent will always be confined in the enclosing continuation.[2] The `control` operator, on the other hand, does not place a delimiter on the captured continuation. Thus, any call to `control` from inside a captured continuation will result in a newly captured continuation which is delimited by the innermost `prompt`; and this `prompt` may lie outside the enclosing continuation.

The difference between static and dynamic control operators can be best shown by an example:

```
(prompt
  (cons
    (control k (cons 1 (k 2)))
    (control j '())))
```

Listing 3.6: Dynamically-extended Continuation.

In this expression, the continuation captured and bound to `k` is `(cons □ (control j '()))`. Applying `k` to `2` results in the evaluation of `(cons 2 (control j '()))`. In this sub-expression, the continuation captured by the second call to `control`, which is bound to `j`, is delimited, not by the enclosing continuation, i.e., `k`, but by `prompt`. Therefore, the entire continuation is discarded and the value `'()` is returned.

Now, if we replace `control`/`prompt` with `shift`/`reset` and evaluate the program again, the continuation captured and bound to `k` is `(cons □ (shift j '()))`. Applying `k` to `2` triggers the evaluation of the expression `(cons 2 (shift j '()))`. This time, however, `j` is bound to the continuation `(cons 2 □)` instead of the one delimited by `reset`. That is, `j`'s extent does not fall outside `k`; and it is this continuation that is discarded, yielding `'()`. This value is in turn passed to the current continuation, which is `(cons 1 □)`, and the final result is `'(1)`.

Obviously, `shift` can be simulated by `control`: we enclose every invocation of a captured continuation in a `prompt`; but it is also possible to go the other way around. Biernacki, Danvy and Shan [5] investigate thoroughly the difference between `control`/`prompt` and `shift`/`reset` described above, and the effects that it has in complicated problems.

[2]An analogy can be drawn with lexical vs. dynamic scoping in functional programming languages.

The last set of delimited control operators we will examine in this thesis was proposed and implemented by Dybvig, Peyton Jones and Sabry [10]. Unlike the delimited control operators presented above, these do not constitute a pair. Instead, the proposal defines the following four operators: `newPrompt`, `pushPrompt`, `withSubCont` and `pushSubCont`. Informally, given terms t_1, t_2 of the call-by-value λ-calculus, the four operators are defined as follows:

`newPrompt`. The `newPrompt` operator creates and returns a new prompt. A prompt is a construct that is used as the delimiter when a continuation is captured. The framework ensures that each successive call to `newPrompt` returns a fresh prompt.

`pushPrompt` t_1 t_2. The `pushPrompt` operator first evaluates t_1, which expects to yield a prompt, and then uses this prompt to delimit the current continuation in the evaluation of t_2. The `pushPrompt` operator generalizes `reset` and `prompt`.

`withSubCont` t_1 t_2. The `withSubCont` operator evaluates t_1 and t_2 which expects to yield a prompt and a function, respectively. Then, it captures the continuation that is delimited by the innermost enclosing `pushPrompt` that corresponds to the same prompt, aborts the current continuation delimited by the same `pushPrompt`, and calls its function argument, passing it a functional abstraction of the captured continuation. Note that the captured continuation does not contain the delimiter whereas when aborting the part of the current continuation, it also aborts `pushPrompt`. The `withSubCont` operator generalizes `shift` and `control`.

`pushSubCont` t_1 t_2. The `pushSubCont` operator evaluates t_1, which expects to be a captured continuation, and composes the current continuation with the result of t_1. Then, it evaluates its second argument t_2 and passes the result to the composed continuation. Put simply, `pushSubCont` is the operator that is used to invoke the captured continuation returned from the evaluation of t_1, passing it the result of t_2.

The following simple example, which is taken from [10], illustrates how the four operators work:

```
((lambda (p)
  (+ 2
    (pushPrompt p
      (if (withSubCont p
            (lambda (k)
               (+ (pushSubCont k #f)
                  (pushSubCont k #t))))
          3
          4))))
(newPrompt))
```

Listing 3.7: Example using the multi-prompt framework.

Here, `newPrompt` is first evaluated, returning a fresh prompt and binding it to `p`. Then, `withSubCont` captures the continuation up to `pushPrompt`'s call, which is associated with the same prompt, and binds the captured continuation to `k`. The captured continuation is `(if □ 3 4)`. Next, the body of the lambda abstraction passed to `withSubCont` is evaluated which is the sum of two continuation calls. This expression can also be regarded as:

```
(+ ((lambda (x) (if x 3 4)) #f)
   ((lambda (x) (if x 3 4)) #t))
```

In other words, `pushSubCont` is used to push the captured continuation, once with `#f` and once with `#t`. The first invocation yields `4` and the second yields `3`, and their sum is passed to the current continuation `(+ 2 □)` which yields `9`.

This set of delimited control operators is more general than all first-class control operators already presented in this thesis. In fact, they form the foundation, based on which, all the other operators can be built. For example, assuming we have created a *top-level* prompt p_0, i.e., a prompt that delimits the continuation that captures the entire computation, we can define an operator `withCont` as follows:

`withCont e = withSubCont` p_0 `(lambda (k) pushPrompt` p_0 `(e k))`

This operator captures the top-level continuation, binds it to `k` and passes it to its argument `e`. Having defined `withCont`, we can easily simulate `call/cc`:

```
call/cc = (lambda (f)
             withCont (lambda (k) pushSubCont k (f (reify k)))
```

where `reify k = (lambda (v) abort (pushSubCont k v))` and `abort e = withCont (lambda (d) e)`. Notice that the definition of `call/cc` is a direct implementation following the semantics given in the previous paragraphs. Similarly, we can define `shift/reset` and `control/prompt`.

In this chapter we explored continuations. We introduced the basic concepts and we considered two notable applications of CPS: definitional interpreters and denotational semantics. We also studied control operators, a control abstraction mechanism of functional programming languages, and we examined their behaviour. Last, we focused on a generalization of traditional control operators, called delimited control operators that are used to manipulate continuations whose extent can be determined by the programmer, and we reviewed several proposals. The last proposal we concerned ourselves with, is a multi-prompt framework that defines four operators. The distinctive property of these operators is that they handle continuations delimited by specific prompts. In other words, there can be multiple prompts in a single program, and the reifier operator can capture a continuation that is delimited not by the innermost delimiter, as is the case with other proposals, but by a corresponding prompt. This feature is proved to be very useful in the simulation of an abstract machine for the call-by-need λ-calculus, as we will see in the following chapter.

Chapter 4

Lazy Pattern Matching and Delimited Continuations

This chapter presents the main contribution of this thesis. We begin by describing the work of Garcia, Lumsdaine and Sabry [10], and their abstract machine that reveals a connection between lazy evaluation and delimited continuations. Next, we extend the abstract machine, by adding two rules, so that it also evaluates user-defined data types and pattern matching expressions following a lazy semantics. Last, we show how these rules can be simulated by an interpreter using delimited control operators.

4.1 An Abstract Machine for a Lazy Language

In their paper, Garcia *et al.* [12] devise an abstract machine for the call-by-need λ-calculus as formalized by Ariola and Felleisen [3]. Instead of using an explicit heap-like store, the machine follows a novel approach to simulate variable bindings with terms: it properly employs evaluation contexts as a replacement of store-based operations. The way the evaluation contexts are used exposes a profound relation with delimited continuations in that when a value of a term, bound to a variable, is needed, the machine restores the part of the context that belongs to the variable's scope.

In this paragraph we present the basic building blocks of the abstract machine. The language that the machine evaluates has the following syntax:

$$t ::= x \mid \lambda x.t \mid t\ t \mid b \mid f\ b \mid \texttt{cons}\ t\ t \mid \texttt{head}\ t \mid \texttt{tail}\ t \mid \langle x, x\rangle$$

with v denoting the set of values:

$$v ::= \lambda x.t \mid b \mid \langle x, x\rangle$$

The language supported by the abstract machine is the untyped λ-calculus augmented with constants b, constant functions f, and *lazy pairs*. A lazy pair, created with `cons`, is a pair whose elements are evaluated lazily. The value $\langle x_1, x_2\rangle$ is a pair that has been constructed with `cons`, and its elements are variables bound to terms. The functions `head` and `tail` are used to extract the elements from a pair. That is, if t is a pair constructed with `cons`, then `head` t extracts t's first element while `tail` t extracts its second element.

An important component of the call-by-need λ-calculus that is related with the notion of shared computation, is that of an *answer*:

$$a ::= v \mid (\lambda x.a)\ t$$

An answer is a syntactic representation of a *binding* (or a *closure*) and as we will see, it is what the abstract machine returns as the result of a computation.

A machine configuration consists of an evaluation context and a term. A configuration of the form $\langle E, t\rangle$ denotes the term $E[t]$. The evaluation of the term starts with an empty context and terminates when the composition of the context with the term yields an answer. The rule that is followed on each step depends primarily on the type of the term, but also on the form of the evaluation context.

The machine uses a predefined set of evaluation contexts, contained in single-element lists, called *frames*. A concatenation of two lists, denoted by $\circ$, replaces the hole of the first context with the second context, e.g., $[E_1] \circ [E_2]$ is equivalent to $[E_1[E_2]]$. On each step, a new frame, whose type depends on the rule executed at a given moment, is pushed onto the top of the evaluation context.

The frames that are used by the abstract machine are the following:

$$[(\lambda x.\Box)\ t]$$

$$[(\kappa x.E)\ \Box]$$

$$[\Box\ t]$$

The first frame is called *binder* and it represents a variable binding. That is, a binder is an evaluation context that represents a function application whose hole is the body of the function. A binder frame is pushed whenever a reduction takes place.

The second frame, which has the form $[(\kappa x.E)\ \Box]$, is called *cont* and it represents a context that waits for an operand which is in turn substituted for x. Here, $\kappa x.E$ denotes a continuation and is equivalent to the (meta-)expression $\lambda x.E[x]$. The former notation, however, does not merely represent a term; it indicates that E has already been explored by the machine. A cont frame is pushed onto the evaluation context whenever a variable is referenced. That is, when the value of a variable x is needed, the machine finds the binder frame that corresponds to x and constructs a cont frame that encloses the context up to that binder. Thus, the machine can focus on the operand part of the binder while it retains the segment that consists of the frames which have been pushed after x's binder.

Finally, a frame of the form $[\Box\ t]$ is called *operand* and it represents a suspended function application that has a standard operand but waits for an operator. The machine pushes an operand frame whenever it handles a function application.

To illustrate how the machine operates, we will go through the steps followed in a simple example such as that of a function application. The evaluation starts with

$$\langle [\,], t_1\ t_2\rangle$$

According to the transition rules, the machine first pushes an operand frame onto the evaluation context and focuses on t_1:

$$\langle [\Box\ t_2], t_1\rangle$$

If we assume that t_1 is already a lambda abstraction, say $\lambda x.t_3$, the machine retrieves t_2 and pushes a binder frame, i.e., $[(\lambda x.\Box)\ t_2]$ onto the context, and focuses on t_3:

$$\langle [(\lambda x.\Box)\ t_2], t_3\rangle$$

Now, while the machine keeps evaluating t_3, it pushes frames onto the context as required. If, at some point, the value of x is needed, the machine will have to search into the context and find the correct binder frame, that is, the frame that corresponds to x.

Obviously, one option would be to start searching the context, frame by frame, until the machine finds the binder frame in question. However, that would be utterly inefficient because the number of frames, which have been pushed after the binder frame, is unknown. In addition, when the binder frame is found, the context up to (but not including) that frame will need to be stored. Ideally, the machine could accomplish these actions in one step.

This is where the abstract machine uses the idea of context delimitation. When it pushes a binder, it also sets a delimiter (or a prompt). If, later on, the value of this variable is needed, then the machine captures the evaluation context, delimited by the prompt of the corresponding binder, and focuses on the binding. This can be best illustrated by the example below:

$$\langle E_1 \circ [(\lambda x.\square)\ t_2] \circ E_2, x \rangle$$

Looking carefully at this configuration we can infer that at some point in the past the reduction rule pushed a binder frame for x and went on to evaluate the rest of the term, while it kept pushing frames as the evaluation proceeded. The part of the context that consists of the frames pushed after x's binder frame is represented by E_2. Now that the value of x is needed, and thereby has to be dereferenced, the corresponding binder frame has to be retrieved. Fortunately, having set a prompt that delimits the context up to that binder allows the machine to capture this part of the context and find the binder in a single step. Once it finds $[(\lambda x.\square)\ t_2]$, the machine pulls the operand t_2 of the binder, it pushes a cont frame $[(\kappa x.E_2)\ \square]$ onto the context and turns its focus on t_2. When it finishes evaluating t_2, it pushes a new binder frame for x, whose operand now is a value, restores E_2 and proceeds with the reduction. Thus, in case x's value is needed again, the machine will find the new binder and thereby will not have to re-evaluate it.

Another key feature of the abstract machine that relies on context delimitation is when it carries out a reduction. More precisely, when the machine is focused on a value, which implies that the term is in normal form, the next thing it does is look into the evaluation context. If the context consists only of binder frames, then the computation terminates. If, on the other hand, the context contains either a cont or an operand frame, then the machine captures the part of the context up to the innermost non-binder frame. Having this segment, which consists only of binder frames, the machine can perform a reduction of the call-by-need λ-calculus. That is, it pushes the value into the binder frames and proceeds with the reduction that depends on the type of the innermost non-binder frame. In this case, however, the machine does not need a new prompt for each reduction; one prompt suffices for all the reductions. This prompt is initialized in the beginning of the program's evaluation, and it is used to extend the captured context every time a cont or an operand frame is pushed.

The transition rules of the abstract machine are divided into four groups: *refocus*, *rebuild*, *need* and *reduce*. The *refocus* set of rules examine the term: if it is a value the machine transitions to *rebuild*, whereas if it is a variable it transitions to the *need* rules. If the term is an application, the machine pushes an operand frame and refocuses on the operator part of the application.

The *rebuild* rules search into the context for the next available redex. That is, when the machine is in the stage of *rebuild*, it pulls the innermost non-binder frame and switches to the *reduce* rules according to the type of the frame.

Last, the machine transitions to the *need* phase when the value of a variable is required. In this case, the *need* rule finds the binder frame that corresponds to the variable in question and refocuses on the operand of the binder.

4.2 Extending the Abstract Machine

In this section we extend the language supported by the abstract machine by adding pattern matching expressions. The syntax of the extended language is defined as follows:

$$\begin{aligned} d &::= \texttt{data}\ \mathit{TConstructor} = \mathit{DConstructor}\ t, \ldots, t \\ t &::= \ \ldots \mid \mathit{DConstructor}\ t_1, \ldots, t_n \mid \texttt{case}\ v\ \texttt{of}\ [p_1 \mapsto e_1, \ldots, p_m \mapsto e_m] \\ v &::= \ \ldots \mid \mathit{DConstructor}\ t_1, \ldots, t_n \end{aligned}$$

In the definitions above, d denotes the set of algebraic data type declarations, where *TConstructor* and *DConstructor* are strings for the type and the data constructor, respectively, as explained in Section 2.3. Having a way to define a data type, we can add two new terms: the first term is a value constructed with a user-defined data type. A value of this type consists of the constructor's name along with a list of terms which may or may not be evaluated. The second term that we add to the language is a `case` expression. Here, v is a value. In this work we consider only the case in which v is restricted to be a constructor value. Furthermore, we denote a list of branches by $[p_1 \mapsto e_1, \ldots, p_n \mapsto e_m]$, where p_i is a pattern with its set of variables $i'_1 \ldots, i'_n$ and e_i is the expression which is evaluated once a pattern match succeeds.

To extend the abstract machine so that it also evaluates `case` expressions we add two new transition rules. The first rule handles constructors. Since we consider them to be values, when the machine is focused on a data constructor, it changes its stage and moves to the *rebuild* rules. (The subscript of the function indicates the group that a rule belongs.) This allows the machine to remain focused on the constructor while it searches into the context to find the nearest non-binder frame and proceed with the reduction. If no such frame is in the context, that is, the evaluation context contains only binder frames, the machine returns a tuple consisting of the binders and the data constructor value, and halts. If a non-binder frame does exist, the machine turns from *rebuild* to *reduce* and performs the reduction.

$$\langle E,\ \mathit{DConstructor}\ t_1, \ldots, t_n\rangle_{\mathit{refocus}} \quad \longmapsto \quad \langle E,\ \mathit{DConstructor}\ t_1, \ldots, t_n\rangle_{\mathit{rebuild}}$$

As a simple example of the rule above, we consider the following term:

$$(\lambda x.\mathit{Constr}\ x)\ \lambda y.y$$

where *Constr* is a data constructor already declared. The machine begins the evaluation with an empty context:

$$\langle [\ ], (\lambda x.\mathit{Constr}\ x)\ \lambda y.y\rangle_{\mathit{refocus}}$$

Since the term is a function application, the machine pushes an operand frame and focuses on the operator:

$$\langle [\square\ \lambda y.y], \lambda x.\mathit{Constr}\ x\rangle_{\mathit{refocus}}$$

Now, it realizes that the term is a value and shifts to *rebuild*, to find the nearest reduction. Once it finds the operand frame pushed in the previous step, it carries out the reduction by pushing a binder frame and then focusing on the body of the lambda abstraction:

$$\langle [(\lambda x.\Box)\ \lambda y.y],\ \mathit{Constr}\ x\rangle_{\mathit{refocus}}$$

Being in *refocus*, the machine invokes our rule for constructors, and it shifts to *rebuild.*

$$\langle [(\lambda x.\Box)\ \lambda y.y],\ \mathit{Constr}\ x\rangle_{\mathit{rebuild}}$$

But at this point, the term is a (constructor) value and the evaluation context contains only binder frames. Thus, the computation terminates. Had the evaluation context contained any non-finder frame, the *rebuild* rule would have pulled the nearest such frame and the machine would have moved to the *reduce* phase.

The next rule handles `case` expressions. This rule states that when the machine reaches a pattern matching expression, it starts comparing the constructor's name with all the patterns of the list, from left to right, until a match succeeds. Once this happens, the machine creates a set of binder frames and pushes this set onto the evaluation context. Then, it focuses on the term of the branch whose match succeeded.

$$\begin{aligned}&\langle E,\ \texttt{case}\ v\ \texttt{of}\ [p_1 \mapsto e_1, \ldots, p_m \mapsto e_m]\rangle_{\mathit{refocus}} \quad \longmapsto \langle E \circ E_1, e_i\rangle_{\mathit{refocus}}\\ &\qquad \text{where } E_1 = ((\lambda i'_1.(\lambda i'_2.(\ldots(\lambda i'_n.\Box)\ t_n)\ldots)\ t_2)\ t_1)\\ &\qquad \text{and } v =_d p_i\end{aligned}$$

Here, $t_1, \ldots, t_n$ are the constructor's elements and $i'_1, \ldots, i'_n$ are pattern variables. Moreover, $=_d$ is a predicate that holds whenever the data constructor, used to create a value, is the same as that of the pattern being matched. The set of binder frames pushed by the transition rule for `case` expressions represents all the bindings of the pattern variables with the constructor's elements. When a variable is referenced during the evaluation of the term of the successful branch, the machine transitions to the *need* rule and evaluates the term with which the pattern variable is bound. By using binder frames for pattern variable bindings we accomplish laziness in pattern matching without further complicating the abstract machine. Put differently, non-strict pattern matching evaluation is achieved using the same rules that delay the evaluation of regular variables in the original language; but instead of pushing one binder frame at a time, which is the case when the machine evaluates a function application, it creates a number of binders, and pushes them all at once. Furthermore, having checked if the constructor matches any of the patterns, the machine does no longer need the name of the constructor and thereby it can safely discard it. It stores in the context its arguments bound with the pattern variables and remains in *refocus*, but now the term being evaluated is the term of the matching branch. If, during the latter term's evaluation, the value of a pattern variable is needed, the *need* rule will find the corresponding frame, it will store the context up to that frame, and eventually, it will evaluate the operand part of the binder.

As an example that illustrates the use of the above rule, we consider the following term:

$$\texttt{case}\ \mathit{Constr}\ (\lambda x.x)\ (\lambda y.y)\ \texttt{of}\ [\mathit{Constr}\ i'_1\ i'_2\ \mapsto\ \mathit{Constr1}\ i'_2\ \lambda z.z]$$

Here, we assume that *Constr* and *Constr1* are already defined and each of them has two elements. Again, the machine starts with *refocus* and an empty context.

$$\langle [\,], \texttt{case}\ \mathit{Constr}\ (\lambda x.x)\ (\lambda y.y)\ \texttt{of}\ [\mathit{Constr}\ i'_1\ i'_2\ \mapsto\ \mathit{Constr1}\ i'_2\ \lambda z.z]\rangle_{\mathit{refocus}}$$

The term that is evaluated triggers the rule for `case` expressions that we added. This rule first attempts to match the constructor with the pattern. The match succeeds and the machine creates two binder frames, one for each of the constructor's elements. Then, it pushes these binder frames onto the context and switches to evaluate the term of the branch:

$$\langle [(\lambda i'_1.\Box)\ \lambda x.x] \circ [(\lambda i'_2.\Box)\ \lambda y.y], \mathit{Constr1}\ i'_2\ \lambda z.z\rangle_{\mathit{refocus}}$$

Now, the machine turns to our first rule (for constructors) and changes to *rebuild*. At this point the term is a value and the context contains only binder frames. That is, we have an answer and therefore, the computation terminates. Note that although in the returned term only i'_2 is referenced, both binders are stored in the context. However, their bound term remains unevaluated until their value is required. Had we demanded that instead of yielding a constructor, the program returned the value of either of the bound terms, we could have used the binders in the context to do so.

The two rules that we introduced extend the abstract machine so that it can also accept user-defined data types and pattern matching expressions, evaluating both kinds of expressions under a lazy strategy. In the following section we show how these rules can be embedded into the interpreter.

4.3 Extending the Interpreter

In their article [12], Garcia *et al.* also present an interpreter that simulates the abstract machine described in Section 4.1. The interpreter translates a program written in a call-by-need language into a call-by-value language with delimited control operators. More precisely, the interpreter takes a λ-term, and returns another λ-term that contains the four delimited control operators introduced by Dyvbig *et al.* in [10]: `newPrompt`, `pushPrompt`, `withSubCont` and `pushSubCont`. Evaluating the latter term with a call-by-value interpreter yields the same result as if the first term had been evaluated under a call-by-need strategy. In the following paragraphs we give an outline of the simulation.

The translation of a program begins by calling `newPrompt` which returns a prompt, typically bound to a variable s. Besides being the *top-level* prompt, s delimits all continuations that represent reductions; these continuations consist only of binder frames. More precisely, before a reduction takes place, the interpreter uses `pushPrompt` to delimit the continuation in which the reduction is performed. Therefore, when the next available reduction is carried out, the interpreter captures a continuation that represents all the bindings related with that reduction. The outcome of this operation is a pair consisting of the captured continuation and the resulting value.

Unlike reductions, for which a single prompt is used in the entire program, binder frames require their own prompt. More specifically, for every binder frame the machine sets a new prompt which then uses to delimit the continuation that is captured when the value of the corresponding variable is needed. This continuation represents the context up to that binder frame which is identified by the *need* rule of the abstract machine. To simulate this behaviour, when the interpreter translates a function application, it first calls `newPrompt` and binds its result to an identifier, which then substitutes for the bound variable inside the body of the operator. Next, it calls `pushPrompt` to delimit the continuation up to the binder frame, suspends the evaluation of the operand and proceeds to evaluate the operator's body. If the value of the variable is needed, `withSubCont` will capture the continuation which is associated with the variable, force the evaluation of the suspended term and finally, it will push its result to the captured continuation.

In the previous section, we introduced two new transition rules: the first rule handles constructor values while the second rule handles `case` expressions. In the current section we extend the interpreter so that it also simulates these two rules. This will allow us to translate pattern matching expressions obeying a non-strict semantics.

The first rule, which is concerned with the evaluation of constructor values, can be simulated following an approach similar to the translation of the other type of values — lambda abstractions. Essentially, the translation function needs to be defined so that when dealing with a constructor value, it captures all the bindings up to the nearest reduction, and returns a representation of an answer. This can be accomplished using the following translation equation:

$$\mathcal{N}[\mathit{DConstructor}\, t_1, \ldots, t_n] \equiv \texttt{withSubCont}\ s\ \lambda k_a.\langle k_a, DConstructor\ t_1, \ldots, t_n\rangle$$

Adopting the original paper's [12] stylistic choices, we denote by $\mathcal{N}[\cdot]$ the translation function; that is, the function that translates a call-by-need term into call-by-value with delimited control operators. According to the above equation, the translation of a constructor value yields a pair whose first element is the continuation delimited by s, and its second element is the value itself. This equation relies on the fact that the translation uses `pushPrompt` to delimit the continuation before every reduction, thus making sure that when the continuation is captured it will contain only bindings.

The second rule, which is concerned with the evaluation of `case` expressions, creates a set of bindings and goes on to evaluate the term of the matching branch. To simulate the latter task we introduce a built-in function, called `matchCase`, that takes two arguments: the first argument is the constructor value passed to `case`, and the second argument is the list of branches. This function returns the term associated with the pattern that matches the constructor. What `matchCase` actually does is perform a textual comparison between the data constructor passed to `case` and each of the patterns of the list. Once it finds a match, `matchCase` returns the term of the branch that succeeded.

The bindings of the rule can be carried out with the translation equation for function applications. The constructor elements are thus successively bound to pattern variables and these bindings surround the result of the call to `matchCase`.

$$\begin{aligned}&\mathcal{N}[\texttt{case}\ v\ \texttt{of}\ [p_1 \mapsto e_1, \ldots, p_m \mapsto e_m]]\\ &\equiv \texttt{let}\ t = \texttt{matchCase}\ v\ [p_1 \mapsto e_1, \ldots, p_m \mapsto e_m]\\ &\quad \texttt{in}\ \mathcal{N}[(((\lambda i'_1. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_n]\end{aligned}$$

Here, $i'_1, \ldots, i'_n$ are pattern variables and $t_1, \ldots, t_n$ are the elements of the constructor passed to `case`. The translation calls `matchCase` with v and the list of branches, and binds the resulting term to t, which then uses as the body of a lambda abstraction. In other words, t is an identifier for e_i where i is such that p_i's data constructor matches with v's data constructor. Furthermore, the bindings $(((\lambda i'_1. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_n$ represent all the binder frames that are created and pushed when the `case` rule of the abstract machine is invoked.

If we expand the translation of the outermost function application we end up with the following expression:

$$\begin{aligned}&\texttt{let}\ t = \texttt{matchCase}\ v\ [p_1 \mapsto e_1, \ldots, p_m \mapsto e_m]\\ &\texttt{in let}\ \langle k_a, v_a\rangle = \texttt{pushPrompt}\ s\ \mathcal{N}[(((\lambda i'_1. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_{n-1}]\\ &\quad \texttt{in pushSubCont}\ k_a\ (\texttt{let}\ x_p = \texttt{newPrompt}\\ &\qquad\qquad\qquad \texttt{in let}\ f_k = \texttt{pushPrompt}\ x_p\ (v_a\ x_p)\\ &\qquad\qquad\qquad\quad \texttt{in}\ f_k\ \lambda().\mathcal{N}[t_n])\end{aligned}$$

The translation first binds the term of the matching branch to t — which is `matchCase`'s work to find it — and executes

$$\begin{aligned}&\mathcal{N}[(((\lambda i'_1. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_{n-1}]\\ &\quad \equiv \texttt{withSubCont}\ s\ \lambda k_a.\langle k_a, \lambda i'_1.\mathcal{N}[(((\lambda i'_2. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_{n-1}]\rangle\end{aligned}$$

When evaluated by a call-by-value interpreter, the expression above returns a pair $\langle k_a, v_a\rangle$, where k_a is a continuation that contains all the binders produced by the translation of

$$\mathcal{N}[(((\lambda i'_1. \ldots \lambda i'_n.t)\ t_1) \ldots)\ t_{n-1}]$$

Each of these binders is represented by a continuation. In general, the translation of each function application of the form $(\lambda i'_j.t)\ t_j$ returns a continuation that represents the binder $(\lambda i'_j.\Box)\ t_j$, which is in turn composed with the continuation that encloses all the binders produced by the evaluation of $\mathcal{N}[t]$. These continuations are all delimited by s.

In addition to the continuation, the pair also contains $v_a = \lambda i'_j.\mathcal{N}[\ldots]$. Once this pair is produced, a new prompt is created to which v_a is applied. This action leads to the replacement of all occurrences of i'_j with the newly created prompt in v_a's body. As we will see below, this prompt, which specifically corresponds to i'_j, is effectively used to delimit the context in which i'_j is visible; that is, it specifies the variable's scope.

Applying $v_a = \lambda i'_j.\mathcal{N}[\ldots]$ to the prompt triggers the simulation of its outer binding using the exact same steps as with the previous binding.

The process continues until all bindings are built. Finally, the translation of the term t, which is returned from `matchCase`, is evaluated inside the binder continuation.

During t's evaluation, if the value of a pattern variable is needed the interpreter will invoke the *need* rule just as it does with ordinary variables. The translated program will use `withSubCont` with the variable itself — which has been replaced with a prompt — as its first argument, and will capture the continuation up to the corresponding binder. Then, it will force the evaluation of the suspended operand of the binder. The result of the bound term's evaluation will be used twice: once as the value passed to the captured continuation and once as an operand to a new binder frame that replaces the old with the unevaluated term. Last, the captured continuation will be composed with the current continuation and the execution will continue with the rest of t.

An example The procedure described above can be best demonstrated by an example. Consider the following program:

$$\texttt{data}\ \mathit{Constr} = \mathit{Constr}\ t\ t$$
$$\texttt{case}\ \mathit{Constr}\ \Omega\ ((\lambda x.x)\ \lambda y.y)\ \texttt{of}\ [\mathit{Constr}\ i'_1\ i'_2\ \mapsto\ i'_2]$$

Here, Ω denotes a non-terminating term. First, we declare an algebraic data type, called *Constr*, that has two elements. Then, we use a value constructed with *Constr* in a simple `case` expression. This expression attempts to match the value with the first (and only) pattern, and if the match succeeds, it returns the second element of the constructor. Moreover, when the value is matched with the pattern both of its elements are bound to the pattern variables; but since only the second element is needed, as opposed to the first element which is never used, the evaluation eventually halts.

The translation begins with the initialization of the program: a new prompt, bound to s — the reduction prompt — is constructed:

```
let $s$ = newPrompt
in pushPrompt $s\,\mathcal{N}[$case $\mathit{Constr}\,\Omega\ ((\lambda x.x)\ \lambda y.y)$ of $[\mathit{Constr}\,i'_1\ i'_2\ \mapsto\ i'_2]]$
```

Having set and pushed the top-level prompt, we use the translation of `case` expressions as follows:

```
let $s$ = newPrompt
in pushPrompt $s$
    let $t$ = matchCase $(\mathit{Constr}\,\Omega\ ((\lambda x.x)\ \lambda y.y))\ [\mathit{Constr}\,i'_1\ i'_2\ \mapsto\ i'_2]$
    in $\mathcal{N}[((\lambda i'_1.\lambda i'_2.t)\ \Omega)\ ((\lambda x.x)\ \lambda y.y)]$
```

The call to `matchCase` in the program above returns the term of the first branch whose pattern matches with the constructor value. Then, we enclose this term in two lambda abstractions, $\lambda i'_1.\lambda i'_2.(\ldots)$, where again, i'_1 and i'_2 are the two pattern variables, and we apply the result to the elements of the constructor value. Next, we translate this term as a typical function application:

```
let $\langle k_a, v_a\rangle$ = pushPrompt $s\,\mathcal{N}[(\lambda i'_1.(\lambda i'_2.i'_2))\ \Omega]$
in pushSubCont $k_a$
    let $x_p$ = newPrompt
    in let $f_k$ = pushPrompt $x_p\ (v_a\ x_p)$
        in $f_k\ \lambda().\mathcal{N}[(\lambda x.x)\ \lambda y.y]$
```

For the sake of simplicity, in the expression above we expand only the outermost function application, i.e., $(\lambda i'_1.[\ldots])\ ((\lambda x.x)\ \lambda y.y)$, but the procedure followed for the second application is identical.

The translation $\mathcal{N}[(\lambda i'_1.(\lambda i'_2.i'_2))\ \Omega]$ yields a pair $\langle k_a, v_a\rangle$, where k_a is a continuation that represents the binder $(\lambda i'_1.\square)\ \Omega$ and $v_a = \lambda i'_2.\mathcal{N}[i'_2]$. Applying v_a to x_p substitutes the prompt stored to x_p for all the occurrences of i'_2 in v_a's body. Note that when this replacement takes places, the evaluation of the translation of the inner application, i.e., $(\lambda i'_1.\lambda i'_2.i'_2)\ \Omega$, has already created another prompt which has been used to replace i'_1 in the body of $\lambda i'_1.\mathcal{N}[\lambda i'_2.i'_2]$ (although there is no occurrence of i'_1 in it).

Once the pair $\langle k_a, v_a\rangle$ is created and v_a is applied to x_p, `pushSubCont` composes k_a with the current continuation and evaluates the translation $\mathcal{N}[i'_2]$, enclosed in a new binder frame $(\lambda i'_2.\square)\ ((\lambda x.x)\ \lambda y.y)$, in the extended continuation. In other words, the evaluation of $\mathcal{N}[i'_2]$ will occur inside the context represented by the composition of $(\lambda i'_1.\square)\ \Omega$ with $(\lambda i'_2.\square)\ ((\lambda x.x)\ \lambda y.y)$. Most importantly, each of these binders is delimited by its *own* prompt.

At this point, the simulation proceeds with the evaluation of $\mathcal{N}[i'_2]$ in which the value of the variable i'_2 is immediately required. Therefore, the translation begins the simulation of the *need* rule.

$$\mathcal{N}[i'_2] \equiv \texttt{withSubCont}\ i'_2\ \lambda k.\lambda f_{th}.[\ldots]$$

This rule uses the variable (which is now a prompt) to capture the context up to — but not including — i'_2's binder. (In our case, this context is empty but it could have equally well been any context.) Then, it stores the captured continuation to k, and returns a function $\lambda f_{th}.[\ldots]$. The function $\lambda f_{th}.[\ldots]$ can be thought of as a simulation of $\kappa x.E$ where E is the part of the context that is represented by k. This function is applied to the suspended term, forces its evaluation, and replaces the binder itself with another; but now the bound term of the new binder is a value.

$$\begin{aligned}
&\lambda f_{th}.\texttt{let}\ \langle k_a, v_a\rangle = \texttt{pushprompt}\ s\ (f_{th}\ ())\\
&\qquad \texttt{in pushSubCont}\ k_a\ (\texttt{let}\ f_k = \texttt{pushPrompt}\ i'_2\ \texttt{pushSubCont}\ k\ \mathcal{N}[v_a]\\
&\qquad\qquad\qquad\qquad\qquad \texttt{in}\ f_k\ \lambda().\mathcal{N}[v_a])
\end{aligned}$$

Last, the current continuation $(\lambda i'_1.(\lambda i'_2.\square)\ \lambda y.y)\ \Omega$ is composed with the captured continuation, and $\square$ is replaced with $\lambda y.y$.

In this chapter we explored the abstract machine of a call-by-need language, as well as its simulating interpreter, of Garcia *et al.* Furthermore, we extended its supported language by adding rules for handling algebraic data types and pattern matching expressions. These rules allow us to evaluate `case` expressions in a way that respects laziness. The new rules do not complicate significantly the abstract machine. Instead, they rely on concepts already defined for the original language. This extension can form the basis for an even more complicated pattern matching language.

Chapter 5

Conclusion

Non-strictness, as prescribed by call-by-name and call-by-need, allows functions to terminate, even when one or more of their arguments do not, as long as the value of these arguments is not needed (or is partly needed) inside the function's body. Pattern matching in non-strict languages behaves similarly, evaluating no more than whatever is required to decide whether its expression argument matches a pattern or not. This makes it an invaluable tool, especially when dealing with infinite data structures.

Abstract machines provide a formal description of a programming language and its features by defining the operational characteristics of the language. In this thesis, we study an interpreter that simulates an abstract machine for the call-by-need λ-calculus. This interpreter translates a program written in a call-by-need language to a call-by-value language endowed with delimited continuations. We extend the language supported by the abstract machine, adding pattern matching expressions, and use the interpreter to simulate these rules.

Delimited continuations play a crucial role in the evaluation of pattern matching expressions. Arguments of the constructors are not evaluated when the matching is performed; instead, they are bound to pattern variables and they are evaluated only when needed. The interpreter treats pattern variables just like ordinary variables. Each pattern variable is assigned a prompt which the evaluator uses to delimit their scope once their values is needed. It also seems that a similar approach could be used to yield an operational semantics and interpret a complete pattern matching language such as Haskell's.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[3] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

[4] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[5] Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Computer Programming*, 60(3):274 – 297, 2006.

[6] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[7] W. Clinger, D.P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In J. Reynolds and M. Nivat, editors, *Algebraic Methods in Semantics: Proceedings of the US-French Seminar on the Application of Algebra to Language Definition and Compilation*, pages 237–250. Cambridge University Press, Cambridge, 1985.

[8] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 151–160, New York, NY, USA, 1990. ACM.

[9] Olivier Danvy and Andrzej Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[10] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, November 2007.

[11] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 180–190, New York, NY, USA, 1988. ACM.

[12] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 153–164, New York, NY, USA, 2009. ACM.

[13] John Hatcliff and Olivier Danvy. Thunks and the Λ-calculus. *Journal of Functional Programming*, 7(3):303–319, May 1997.

[14] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4(2):157 – 176, 1987.

[15] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM.

[16] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[17] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference*

on History of Programming Languages, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[18] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989.

[19] Martin Odersky. Programming in scala. 2007.

[20] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[21] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[22] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[23] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[24] Chung-chieh Shan. Shift to control. In Olin Shivers and Oscar Waddell, editors, *Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming*, pages 99–107, Snowbird, Utah, September 22, 2004. Technical report TR600, Department of Computer Science, Indiana University.

[25] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, USA, 1977.

[26] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Programming Research Group Technical Monograph PRG-11, Oxford Univ. Computing Lab., 1974. Reprinted in *Higher-Order and Symbolic Computation*, vol. 13 (2000), pp. 135–152.

[27] Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: A interpreter for extended lambda calculus. *Higher Order and Symbolic Computation*, 11(4):405–439, December 1998.

[28] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[29] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual.* Network Theory Ltd., 2011.