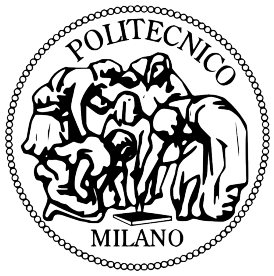
POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria



First-Class Continuations on the Java Virtual Machine: An Implementation within the Kawa Scheme Compiler

Relatore: Prof. Matteo Pradella

Tesi di Laurea di:
Andrea Bernardini
matricola 786244

Anno Accademico 2014-2015

To my family

Abstract

The widespread diffusion of the Java technology has encouraged the birth of new programming languages on the Java Virtual Machine, languages that brings new features to the Java environment, most of which taken from the functional paradigm. Kawa is an implementation of the programming language Scheme on the Java Virtual Machine. As a Scheme it provides a functional style of programming, dynamic typing, and meta-programming facilities. However, being the Java Virtual Machine devoid of stack manipulation primitives, Kawa lacks of one of the most peculiar Scheme features: First-class continuations.

This dissertation describes an implementation of the `call/cc` control operator in the Kawa compiler. In particular it shows how the exception handling feature, common to many programming languages, can be exploited to implement first-class continuations in an environment without stack manipulation primitives, and how this can be realised in a real compiler. This thesis also shows how first-class continuations and control operators like `call/cc` can be used to introduce concurrency features and to implement new control flow constructs in programming languages.

Sommario

La grande diffusione della tecnologia Java ha favorito la nascita di nuovi linguaggi di programmazione per la Java Virtual Machine, linguaggi che offrono nuove funzionalità all'ecosistema Java, la maggior parte dei quali ispirati dal paradigma di programmazione funzionale. Kawa è una implementazione del di linguaggio di programmazione Scheme pel la Java Virtual Machine. Come ogni implementazione di Scheme esso fornisce uno stile funzionale di programmazione, tipizzazione dinamica, e strumenti per la metaprogrammazione. Tuttavia, essendo la Java Virtual Machine priva di primitive per la manipolazione diretta dello stack, Kawa manca di una delle caratteristiche più peculiari di Scheme: le continuazioni di prima classe.

Questa tesi descrive un'implementazione dell'operatore di controllo `call/cc` all'interno del compilatore Kawa. In particolare si mostra come la gestione delle eccezioni, comune a molti linguaggi di programmazione, può essere sfruttata per implementare le continuazioni di prima classe in un ambiente senza primitive di manipolazione dello stack, e come questo può essere messo in pratica in un vero compilatore. Si mostra anche in questa trattazione come le continuazioni di prima classe e gli operatori di controllo come la `call/cc` possono essere usate per introdurre concorrenza e nuovi strutture di controllo nei linguaggi di programmazione.

Acknowledgements

I would like to thank the people who made this thesis possible.

I would like to thank my supervisor Prof. Matteo Pradella, for inspiring me with his course on programming languages, and then for giving me the opportunity to work on this thesis.

Special thanks are given to Per Bothner, the author and project leader of Kawa, for guiding me in the world of compiler programming. Without his help, this thesis would not have been possible.

My deepest gratitude goes to my family, for the continuous support during these years at university, and to my friends, for making this years unforgettable.

Table of Contents

List of Figures

Chapter 1

Introduction

> *"Programming languages are not just technology, but what programmers think in. They're half technology and half religion."*
>
> Paul Graham, Beating the Averages

Context

Functional programming

It is well known that the modern computers are not improving their performance like in the past decades, because frequency scaling, for silicon, has reached a limit. For this reason, processors manufacturers increase the potential productivity of their products by adding cores [1].

This implies that to benefit most from this architecture, programs have to be parallellized. But parallel programming is quite harder than sequential programming, due to several new challenges it brings. *Functional programming* (FP) helps to get rid of some of these challenges, and it has recently risen in importance because it is well suited for parallel, concurrent and event-driven (or "reactive") programming, thanks to the use of immutable variables and functions without side effects. The learning curve for functional programming is often steep, but parallel and concurrent programming with imperative languages is not intuitive and its learning curve might be even steeper.

Functional programming is often used in synergy with other programming paradigms, since the world is made of stateful objects, while FP uses a mainly stateless computation model. FP has ways to model state, but there is an essential mismatch in a stateless model trying to represent a stateful world.

However, there are several programming problems in the world that are easy to map to the FP model. Problems involving concurrency, parallelism, large data sets and multi-processing.

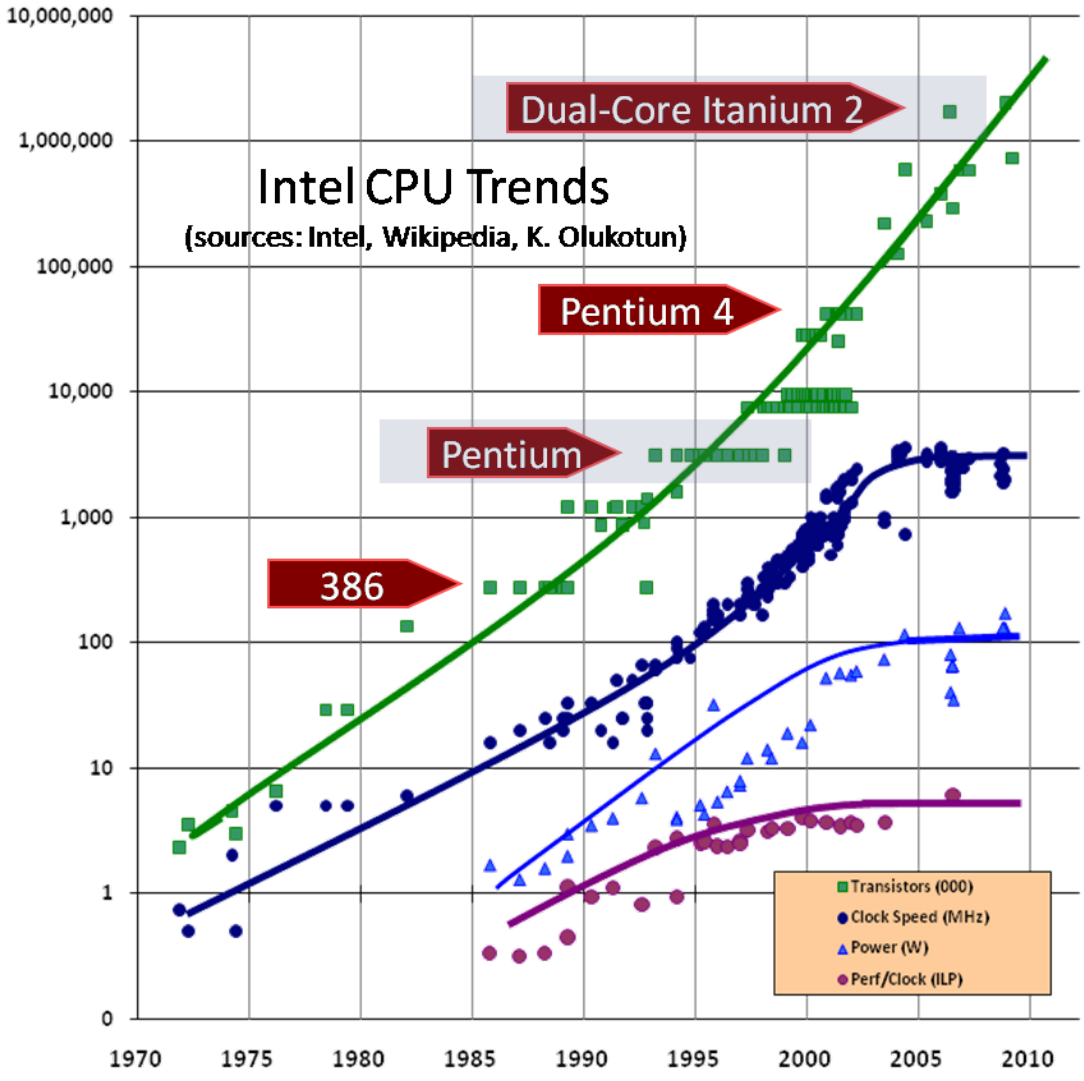


Figure 1.1: Intel CPU Trends [1]

Java

Java is a general-purpose programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any *Java Virtual Machine* (JVM) regardless of computer architecture. As of 2015, Java is one of the most popular programming languages in use [2] (see Figures 1.2 and 1.3). Java was originally developed by James Gosling at Sun Microsystems and released in 1995. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them [3].

The reference implementation Java compilers, virtual machines, and class libraries were open-sourced in May 2007 under the GNU General Public License.

Jun 2015	Jun 2014	Change	Programming Language	Ratings	Change
1	2	↑	Java	17.822%	+1.71%
2	1	↓	C	16.788%	+0.60%
3	4	↑	C++	7.756%	+1.33%
4	5	↑	C#	5.056%	+1.11%
5	3	↓	Objective-C	4.339%	-6.60%
6	8	↑	Python	3.999%	+1.29%
7	10	↑	Visual Basic .NET	3.168%	+1.25%
8	7	↓	PHP	2.868%	+0.02%
9	9		JavaScript	2.295%	+0.30%
10	17	⇈	Delphi/Object Pascal	1.869%	+1.04%
11	-	⇈	Visual Basic	1.839%	+1.84%
12	12		Perl	1.759%	+0.28%
13	23	⇈	R	1.524%	+0.85%
14	-	⇈	Swift	1.440%	+1.44%
15	19	⇈	MATLAB	1.436%	+0.66%
16	13	↓	Ruby	1.359%	-0.03%
17	26	⇈	PL/SQL	1.229%	+0.74%
18	31	⇈	COBOL	0.948%	+0.54%
19	34	⇈	ABAP	0.849%	+0.49%
20	18	↓	Pascal	0.846%	+0.04%

Figure 1.2: TIOBE Index for June 2015 [2]

Programming Language	2015	2010	2005	2000	1995	1990	1985
C	1	2	1	1	2	1	1
Java	2	1	2	3	-	-	-
Objective-C	3	12	39	-	-	-	-
C++	4	4	3	2	1	3	11
C#	5	5	9	8	-	-	-
PHP	6	3	4	25	-	-	-
Python	7	6	8	23	21	-	-
JavaScript	8	8	10	6	-	-	-
Visual Basic .NET	9	-	-	-	-	-	-
Perl	10	7	5	4	7	17	-
Pascal	16	14	41	13	3	9	5
Lisp	25	16	14	7	6	4	2
Fortran	29	24	15	18	4	2	4
Ada	30	26	16	16	5	8	3

Figure 1.3: Positions of the top 10 programming languages of many years back. [2]

Java 8

Starting from release 8, Java supports aspects of functional programming. Two core concepts introduced in Java 8 are *lambda expressions* and *functional interfaces* [4].

A lambda expression is an anonymous function that can be declared with a comma separated list of the formal parameters enclosed in parentheses, an arrow token (->), and a body. Data types of the parameters can always be omitted, as can the parentheses if there is only one parameter. The body can consist of a single statement or a statement block.

Syntax:

```
(arg1, arg2...) -> { body }

(type1 arg1, type2 arg2...) -> { body }
```

Examples:

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }

() -> { return 2.7182 };
```

In Java, lambda expressions are represented as objects, and so they must be bound to a particular object type known as a functional interface. A functional interface

is an interface that defines exactly one abstract method. An extremely valuable property of functional interfaces is that they can be instantiated using lambdas.

An example of a functional interface is `java.lang.Runnable`. It has only one method void `run()` declared. Before Java 8, anonymous inner classes were used to instantiate objects of functional interface. With Lambda expressions, this can be simplified.

Each lambda expression can be implicitly assigned to one functional interface. For example we can create `Runnable` interface's reference from lambda expression like below:

```
Runnable r = () -> System.out.println("running");
```

This type of conversion is automatically handled by the compiler when we dont specify the functional interface. For example:

```
new Thread(
    () -> System.out.println("running")
).start();
```

In above code, compiler automatically deduced that lambda expression can be casted to Runnable interface from `Thread` class's constructor signature `public Thread(Runnable r) { }`.

Few examples of lambda expressions and their functional interface:

```
Consumer<Integer>  c = (int x) -> { System.out.println(x) };

BiConsumer<Integer, String> b = (Integer x, String y)
                                  -> System.out.println(x + y);

Predicate<String> p = (String s) -> { s == null };
```

With the addition of Lambda expressions to arrays operations, Java introduced a key concept into the language of *internal iteration*. Using that paradigm, the actual iteration over a collection on which a Lambda function is applied is now carried out by the core library itself [5]. An relevant possibility opened by this design pattern is to enable operations carried out on long arrays (such as sorting, filtering and mapping) to be carried out in parallel by the framework. For example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

// old way
for (int number : numbers) {
    System.out.println(number);
```

```
}

// new way
numbers.forEach(value -> System.out.println(value));
```

In Java 8 it is also possible to reference both a static and an instance method using the new `::` operator:

```
numbers.forEach(System.out::println);
```

Passing a lambda expression to another function allows to pass not only values but also behaviours and this enables to project more generic, flexible and reusable API. For instance declaring the following method:

```
public void evaluate(List<integer> list,
                     Predicate<integer> predicate) {
    for(Integer n: list)  {
        if(predicate.test(n)) {
            System.out.println(n + " ");
        }
    }
}
```

we can use the `Predicate` functional interface to create a test and print the elements that pass the test:

```
System.out.println("Print all numbers:");
evaluate(numbers, (n)->true);

System.out.println("Print even numbers:");
evaluate(numbers, (n)-> n%2 == 0 );

System.out.println("Print odd numbers:");
evaluate(numbers, (n)-> n%2 == 1 );
```

Java 8 brings to developers another interesting feature from functional programming: *Streams*, that is, lazy *evaluation*. Streams are a new abstraction that allows to process data in a declarative way:

```
System.out.println(
    numbers.stream()
        .filter(Lazy::isEven)
        .map(Lazy::doubleIt)
        .filter(Lazy::isGreaterThan5)
        .findFirst()
);
```

You can create a Stream from any Collection by invoking the `stream()` method on it. A Stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand. They consume from a data-providing source such as collections, arrays, or I/O resources and support common operations, such as filter, map, reduce, find, match, sorted. Furthermore, many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline, enabling also certain optimisations.

The Java Virtual Machine

A Java Virtual Machine (JVM) is an abstract computing machine defined by a specification. The specification formally describes what is required of a JVM implementation. Having a single specification ensures all implementations are interoperable. A JVM implementation is a software platform that meets the requirements of the JVM specification in a compliant and preferably performant manner [6].

One of the main goals of Java design is portability, and Java is indeed platform independent. That is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to architecture-specific machine code. Java bytecode instructions are analogous to machine code, but they are intended to be executed by a virtual machine written specifically for the host hardware. Moreover, Just-in-Time (JIT) compilers were introduced from an early stage that compile bytecodes to machine code during runtime. Thus a JVM is platform dependent, because it must convert Java bytecode into machine language which depends on the architecture and operating system being used. End users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a web browser for Java applets [3].

The Oracle Corporation, which owns the Java trademark, distributes the Java Virtual Machine implementation HotSpot together with an implementation of the Java Class Library under the name Java Runtime Environment (JRE).

JVM based Languages

The JVM is not only for Java. Several hundred JVM programming languages are available to be run on it. These languages ultimately compile to bytecode in class files, which the JVM can then execute.

Some JVM languages include more features than Java and aim to let developers write code in a more concise way. Features like collection literals, pattern matching, and a more sophisticated type inference were the motivation for languages such as Scala, Groovy, Xtend, Ceylon, Kotlin, and Fantom [7].

Then there are existing languages that were ported to the JVM. Python, Erlang, Ruby, Scheme and Javascript, for instance, all have an implementation targeting

the JVM (respectively Jython, Erjang, JRuby, Kawa and Rhino). Another popular language ported to the JVM is Clojure, a dialect of Lisp with an emphasis on functional and concurrent programming [6].

Many less-known JVM languages implement new research ideas, are suited only for a specific domain, or are just experimental.

Scheme

Scheme is a dialect of the computer programming language Lisp. It follows a minimalist design philosophy that specifies a small standard core accompanied by powerful tools for meta-programming.

Scheme was created during the 1970s at the MIT AI Lab by Guy L. Steele and Gerald Jay Sussman. It was the first dialect of Lisp to choose lexical scope and the first to require implementations to perform tail-call optimisation. It was also one of the first programming languages to support first-class continuations [8].

Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation [9].

The storage required to hold the contents of an object is dynamically allocated as necessary and retained until no longer needed, then automatically deallocated, typically by a garbage collector. Simple atomic values, such as small integers, characters, booleans, and the empty list, are represented as primitive types and thus incur no allocation or deallocation overhead [9].

Scheme is *homoiconic*, i.e, programs share a common representation with Scheme data structures. As a result, any Scheme program has an internal representation as a Scheme object. For example, variables and syntactic keywords correspond to symbols, while structured syntactic forms correspond to lists. This representation is the basis for the syntactic extension facilities provided by Scheme for the definition of new syntactic forms. It also facilitates the implementation of interpreters, compilers, and other program transformation tools [9].

In Scheme, a procedure definition may appear within another block or procedure, and the procedure may be invoked at any time thereafter, even if the enclosing block has completed its execution. To support lexical scoping, a procedure carries the lexical context (environment) along with its code.

Furthermore orover, Scheme provides anonymous procedures. Indeed procedures are first-class data objects like strings or numbers, and variables are bound to procedures in the same way they are bound to other objects.

The Scheme language is standardized in the Revised^{n} Report on the Algorithmic Language Scheme (RnRS), where the ^{n} indicates the revision number. The last

report is R7RS, released in 2013.

Scheme basics

Scheme syntax is essential, it provides a minimal set of special forms: define, quote, lambda, cond, let/let*

`define` is used to define new names.

```
(define x 10)
(define square (lambda (x) (* x x)))
```

`quote` prevents the argument to be evaluated as an expression, returning it as literal data (symbols or lists).

```
(quote hi!)           => hi!
(quote (1 2 3))         => (1 2 3)

; the tick-mark ' is syntactic sugar
'(1 2 foo bar)          => (1 2 foo bar)
```

`lambda` is used to create anonymous functions.

```
(lambda (x) (* x 10))                    ; anonymous function
(define times10 (lambda (x) (* x 10))) ; named the function now
```

`cond` is a general conditional.

```
(cond
  ((eq? 'foo 'bar) 'hello)
  ((= 10 20) 'goodbye)
  (else 'sorry))                  => sorry
```

`let` is used to declare/use temporary variables.

```
(let ((x 10)
      (y 20))
  (+ x y))
```

Built-in types are integers, rationals, floats, characters, strings, booleans, symbols, lists, and vectors. A set of built-in functions we can use on these types:

```
;; arithmetic:  +, -, *, /
;; relational: <, <=, >, >=, =
(+ 1 2)                    => 3
(= 1 2)                    => #f ; '=' is for numbers
```

Equality and identity tests:

```
(eq? 'hello 'goodbye)      => #f ; eq? is an identity test
(eq? 'hello 'hello)        => #t
(eq? '(1 2) '(1 2))        => #f
(define foo '(1 2))
(define bar foo)
(eq? foo bar)              => #t
(equal? foo bar)           => #t ; equality: they look the same
(equal? foo '(1 2))        => #t
```

Being a dialect of Lisp, Scheme provides a set of built-in functions for List manipulation: cons, car, and cdr.

```
;; Three equivalent ways to create the list (1 2 3),
;; calling it foo
(define foo '(1 2 3))
(define foo (cons 1 (cons 2 (cons 3 ()))))
(define foo (list 1 2 3))

;; list precessing
(null? '(1 2))             => #f
(null? ())                 => #t
(car '(1 2))               => 1
(cdr '(1 2))               => (2)
```

Iteration via recursion:

```
;; Exponentiation function x^n
(define (expt x n
  (if (= n 0)
      1
      (* x (expt x (- n 1))))))

;; List length
(define (length lst
  (if (null? lst)
      0
      (+ 1 (length (cdr lst))))))
```

It is straightforward to create and use higher order functions. Indeed functions are first-class in Scheme, they can be passed as arguments to other functions:

```
(define compose
  (lambda (f g x)
    (f (g x))))

(compose even? (lambda (x) (- x 1)) 10)   => #f

;; takes a function and applies it to every element of a list
(define (map f lst)
  (let loop ((newlst lst))
    (cond ((pair? newlst)
      (cons (f (car newlst)) (loop (cdr newlst))))
     ((null? newlst)
      '())
     (else
      (error "second argument is not a list:"  lst)))))

(map even? '(1 2 3 4))        => (#f #t #f #t)
```

Continuations

Computer programs usually control the flow of execution via procedure calls and returns; a stack of frames is how high-level programming languages keep track of the point to which each active subroutine should return control when it finishes executing. However, to solve real-world problems, procedure call and primitive expressions are not enough. Thus most high-level programming languages also provide other control-flow primitives, like conditionals, loops, and exception handling.

Scheme also supports *first-class continuations*. A continuation is a Scheme function that embodies "the rest of the computation". The continuation of any Scheme expression determines what is to be done with its value. This continuation is always present, in any language implementation, since the system is able to continue from each point of the computation. Scheme provides a mechanism for capturing this continuation as a closure. The obtained continuation can be used to continue, or resume, the computation from the point it was captured, whether or not the computation has previously completed. This is useful for nonlocal exits in handling exceptions, or in the implementation of complex control structures such as coroutines or generators [10].

Considering a computation such as `(* (+ 2 4) (+ 1 6))`, there are several continuations involved. The continuation for `(+ 2 4)` can be expressed in this way: take this value (6), keep it aside; now add one and six, take the result and multiply it with the value we had kept aside; then finish. The continuation for `(+ 1 6)` means:

take this value, multiply it with the value (6) that was previously kept aside; then finish. Notice in particular how the result of `(+ 2 4)` is part of the continuation of `(+ 1 6)`, because it has been calculated and kept aside. Continuations are not static entities that can be determined at compile time: they are dynamic objects that are created and invoked during program execution.

Using the syntactic form `call-with-current-continuation` (usually abbreviated `call/cc`), a program can obtain its own continuation. This continuation is a Scheme closure that may be invoked at any time to continue the computation from the point of the `call/cc`. It may be invoked before or after the computation returns; it may be invoked more than one time.[1]

The standard idiom for `call/cc` has an explicit lambda term as its argument:

```
(call/cc (lambda (current-continuation)
  body))
```

During the execution of the expression body, the variable current-continuation is bound to the current continuation. If invoked, current-continuation immediately returns from the call to `call/cc`, and `call/cc` returns whatever value was passed to current-continuation.

When applied to a function `f`, `call/cc` captures and aborts the entire continuation `k`, reinstate a copy of `k`, and applies `f` to `k`.

Consider a first example:

```
(call/cc
  (lambda (k)
    (k 42)))
```

This applies `call/cc` to the function `(lambda (k) (k 42))`, which is called with argument `k`, the current continuation. Being the body of the function `(k 42)`, the continuation is thrown the value 42. This makes the `call/cc` return the value 42. Hence, the entire expression evaluates to 42.

Now consider

```
(call/cc
  (lambda (k)
    (+ (k 42) 100)))
```

In this case, the function throws the value 42 to the continuation, but there is another computation afterwards. That computation has no effect, because when a continuation is invoked with a value, the program reinstates the invoked continuation, and the continuation which was going to take a value `x` and perform `(+ x 100)` has been aborted. The result is still 42.

On the other hand, consider

[1] For more explanation and examples on continuations see [11–13].

```
(call/cc
  (lambda (k) 42))
```

Here, the function applied by `call/cc` does not make use of the current continuation. It performs a real return, with the value 42.

Actually, although a continuation can be called as a procedure, it is not a real function, which takes a value and returns another. An invoked continuation takes a value and does everything that follows to it, never returning a value to the caller.

As an other example, consider the following code:

```
(display
    (call/cc (lambda (k)
          (display "This is executed.\n")
          (k "Value passed to the continuation.\n")
          (display "But not this.\n"))))
```

it will display:

```
This is executed.
Value passed to the continuation.
```

An interesting feature of first-class continuations is that the continuation may still be called even after the call to call/cc is finished. When applied to a value `v`, a continuation `k` aborts its entire execution context, reinstates `k` as the current entire continuation, and returns the value `v` to the continuations `k`, which is "waiting for a value" in order to perform some computation with it. In some Scheme implementations, the value passed to a continuation can be a void one.

For example, the following causes an infinite loop that prints `goto start` forever:

```
(let ((start #f))
  (if (not start)
    (call/cc (lambda (cc)
               (set! start cc))))

  (display "goto start\n")
  (start))
```

Delimited Continuations

Continuations captured by `call/cc` is the whole continuation that includes all the future computation. In some cases, we want to manipulate only a part of computation. This is possible with a kind of continuations called *delimited* or *composable* continuations [14].

A continuation is delimited when it produces an intermediate answer rather than the final outcome of the entire computation. In other words, a delimited continuation is a representation of the "rest of the computation" from the current computation up to a designated boundary. Unlike regular continuations, delimited continuations return a value, and thus may be reused and composed [15].

Various operators for delimited continuations have been proposed in the research literature, such as `prompt` and `control`, `shift` and `reset`, `cupto`, `fcontrol`, and others [16]. In this introduction we will consider only the `shift` and `reset` operators.

The `reset` operator sets the limit for the continuation while the `shift` operator captures or reifies the current continuation up to the innermost enclosing `reset`. The `shift` operator passes the captured continuation to its body, which can invoke, return or ignore it. Whatever result that `shift` produces is provided to the innermost `reset`, discarding the continuation in between the `reset` and `shift`. The continuation, if invoked, effectively reinstates the entire computation up to the `reset`. When the computation is completed, the result is returned by the delimited continuation [17]. For example, consider the following snippet in Scheme:

```
(* 2 (reset (+ 1 (shift k (k 5)))))
```

The `reset` delimits the continuation that `shift` captures. When this code is executed, the use of `shift` will bind `k` to the continuation `(+ 1 [])` where `[]` represents the part of the computation that is to be filled with a value. This is exactly the code that surrounds the `shift` up to the `reset`. Since the body of `shift` immediately invokes the continuation, the previous expression is equivalent to the following:

```
(* 2 (+ 1 5))
```

Once the execution of the `shift`'s body is completed, the continuation is discarded, and execution restarts outside `reset`. For instance:

```
(reset (* 2 (shift k (k (k 4)))))
```

invokes `(k 4)` first, which produces 8 as result, and then `(k 8)`, which returns 16. At this point, the `shift` expression has terminated, and the rest of the `reset` expression is discarded. Therefore, the final result is 16.

Kawa

Kawa is a language framework written in Java that implements an extended version of the programming language Scheme. It provides a set of Java classes useful for implementing dynamic languages, such as those in the Lisp family. Kawa is also an

implementation of almost all of R7RS Scheme (First-class continuations being the major missing feature), and which compiles Scheme to the bytecode instructions of the JVM [18]. The author and project leader of Kawa is Per Bothner, who started its development in 1996.

Kawa gives run-time performance a high priority. The language facilitates compiler analysis and optimisation, and most of the time the compiler knows which function is being called, so it can generate code to directly invoke a method. Kawa also tries to catch errors at compile time.

To aid with type inference and type checking, Kawa supports optional type specifiers, which are specified using two colons. For example:

```
(define (add-int x::int y::int) :: String
    (String (+ x y)))
```

This defines a procedure add-int with two parameters: x and y are of type Java `int`; the return type is a `java.lang.String`.

The Kawa runtime start-up is quite fast for a language based on the JVM. This allows Kawa to avoid using an interpreter. Each expression typed into the REPL is compiled on-the-fly to JVM bytecodes, which may be compiled to native code by the just-in-time (JIT) compiler.

Kawa Scheme has several extensions for dealing with Java objects. It allows to call methods of Java objects/classes, create objects and implement classes and interfaces.

For example, the following is Kawa code for an instance of a anonymous class:

```
(object (<java.lang.Runnable>)
  ((run) <void>
   (display "running!\n")))
```

Here a simple class definition:

```
(define-simple-class Person ()
  (last ::String)
  (first ::String)
  ((*init* f l)
   (set! first f)
   (set! last l))
  ((sayHello)
   (display "Hello ")
   (display (string-append first
                           " "
                           last
```

```
                                "!\n"))))

(let ((p (Person "Alyssa" "P. Hacker")))
  (p:sayHello)) ; => Hello Alyssa P. Hacker!
```

Thesis Contributions

My main contribution is an implementation of `call/cc` in a Scheme compiler targeting the JVM. The only other Scheme implementations targeting the JVM are SISC, which is an heap based interpreter, and Bigloo, which is a compiler but does not support continuations in the JVM back-end. Scala implements a different type of control operator, `shift` and `reset`. Although Ruby has `callcc`, JRuby does not support it.

I address the problem of providing a control operator that copies the stack in an environment that prevents direct stack manipulation. Unlike other solutions proposed to implement continuations on the JVM, we perform a transformation on the syntax tree produced by Kawa, instead of a transformation at the bytecode level. This make our transformation independent of the JVM version.

I present a variant of generalised stack inspection, described by Pettyjohn et al., as an extension of the Kawa compiler. The transformation is global, thus has been developed as an optional compiler pass, to avoid adding overhead to programs that do not use continuations.

Outline

The following chapters are organized as follows. Chapter 2 provides a survey of related work. It discusses common techniques for implementing `call/cc` present in literature. Then it also compares different approaches to implement first-class continuations on the JVM.

Chapter 3 presents the issues in delivering first-class continuations on the JVM. It describes the details of the code transformation technique employed to enable the capture and resume of first-class continuations.

Chapter 4 demonstrates the viability of the design by providing an implementation of the entire transformation.

Chapter 5 shows how the proposed implementation can be used to add debugging facilities to Kawa, and to implement new control flow constructs.

Chapter 6 provides a performance evaluation and discusses some issues related this approach. The advantages and limitations of this approach are also discussed in detail.

Finally, Chapter 7 summarizes the contributions of this thesis and discusses possible future work.

Chapter 2

State of the art

"Objective reality is a synthetic construct, dealing with a hypothetical universalization of a multitude of subjective realities."

Philip K. Dick, The Electric Ant

Stack-based implementation techniques for first-class continuations

The most common approach to implement first-class continuations is to use a stack-based execution architecture and to reify the current continuation by making a copy of the stack, which is reinstated when the continuation is invoked. This is the approach taken by many language implementations that are in direct control of the runtime system. This section describes the most used implementation strategies for first class continuations.

The garbage-collection strategy

The simplest strategy for ensuring that continuations have unlimited extent is to allocate them in the heap and to rely on garbage collection to recover their storage. This is called the gc strategy. The gc strategy is not a zero-overhead strategy and it is optimised for programs in which every continuation frame is captured. Few real programs capture all continuations, however, so the gc strategy may not perform as well as a zero-overhead strategy. The most important indirect cost of the gc strategy is that the compiler must allocate a separate continuation frame for each non-tail call, unless the compiler can prove that the continuation will not be captured during the non-tail call. The gc strategy also suffers more cache misses than the other strategies described in this section [19].

The spaghetti strategy

This is a variation of the gc strategy. The spaghetti stack is in effect a separate heap in which storage is reclaimed by reference counting rather than garbage collection. Though complex, the spaghetti stack was at one time more efficient than using a gc strategy with a non-generational garbage collector, because the spaghetti stack's storage management is optimised to support procedure call, return, and a host of related operations. When all frames have dynamic extent, the spaghetti stack behaves as a conventional stack. When a fast garbage collector is available, the spaghetti strategy is probably slower than the gc strategy. Moreover captures and throws require updating the reference counts, thus appears that the gc strategy should always perform better than the spaghetti strategy [19].

The heap strategy

In the heap strategy, a one-bit reference count in each frame indicates whether the frame has been captured. Continuation frames are allocated in a garbage-collected heap, as in the gc strategy, but a free list of uncaptured frames is also used. When a frame is needed by a procedure call, it is taken from the free list unless the free list is empty. If the free list is empty, then the frame is allocated from the heap. When a frame is returned through, it is linked onto the free list if its reference count indicates that it has not been captured. Otherwise it is left for the garbage collector to reclaim. The heap strategy is not a zero-overhead strategy and it is most practical if all continuation frames are the same size; otherwise multiple free lists may be required. This is an indirect cost of the heap strategy. Another indirect cost is that, like the gc strategy, the heap strategy makes it difficult to reuse a continuation frame for multiple non-tail calls [19].

The stack strategy

In the stack strategy, the active continuation is represented as a contiguous stack in an area of storage called the stack cache. Non-tail calls push continuation frames onto this stack cache, and returns pop frames from the stack cache, just as in an ordinary stack-based implementation. When a continuation is captured, however, a copy of the entire stack cache is made and stored in the heap. When a continuation is thrown to, the stack cache is cleared and the continuation is copied back into the stack cache. A first-class continuation thus resides in the heap, but is cached in the stack cache whenever it is the active continuation. The stack strategy is a zero-overhead strategy. Capturing, recapturing, and throwing to a continuation take time proportional to the size of the continuation. An indirect cost of the stack strategy is introduced by the fact that it repeatedly copies the same continuation from the stack cache to the heap. This can increase the asymptotic storage space required. The stack strategy prevents a compiler from allocating storage for mutable variables within a continuation frame, because there are other

copies of it. Mutable variables must generally be allocated in registers or in the heap, that is another indirect cost of the stack strategy [19].

The chunked-stack strategy

By maintaining a small bound on the size of the stack cache, and copying portions of the stack cache into the heap or back again as the stack-cache overflows and underflows, the chunked-stack strategy reduces the worst-case latency of captures and throws. This strategy works well with generational garbage collection because limiting the size of the stack cache limits the size of the root set that the garbage collector must scan on each garbage collection. The portion of the continuation that resides in the heap will be scanned only when its generation is collected. The chunked-stack strategy is a zero-overhead strategy, because the cost of stack-cache overflows and underflows is usually negligible. On the other hand, the chunked-stack strategy requires a stack cache that is large enough to avoid stack-cache overflows and underflows, that degrade performance [19].

The stack/heap strategy

The stack/heap strategy is similar to the stack strategy. All continuation frames are allocated in the stack cache. When a continuation is captured, however, the contents of the stack cache are moved into the heap and the stack cache is cleared. When a continuation is thrown to, the new active continuation is left in the heap and the stack cache is cleared; this can be done in constant time. Since the current continuation may reside in either the stack cache or in the heap, each procedure return must test to see whether the frame should be popped off the stack cache. The stack/heap strategy makes throwing and recapturing a previously captured continuation very fast. A disadvantage of the stack/heap strategy is that it prevents the compiler from reusing a single continuation frame for multiple non-tail calls [19].

The incremental stack/heap strategy

The incremental stack/heap strategy is a variation of the stack/heap strategy: When returning through a continuation frame that isn't in the stack cache, a trap occurs and copies the frame into the stack cache. The trap can be implemented by maintaining a permanent continuation frame at the bottom of the stack cache. This frame's return address points to system code that copies one or more frames from the heap into the stack cache, and immediately returns through the first of those continuation frames. The incremental stack/heap strategy is a zero-overhead strategy, with the same calling sequence as the stack strategy. Since the incremental stack/heap strategy copies frames from the heap into the stack cache, mutable variables cannot be kept within a continuation frame [19].

The Hieb-Dybvig-Bruggeman strategy

A variation of the incremental stack/heap strategy that uses multiple stack segments that are allocated in the heap. The stack segment that contains the current continuation serves as the stack cache. When the stack cache overflows, a new stack cache is allocated and linked to the old one. Stack-cache underflow is handled by an underflow frame, as in the incremental stack/heap strategy. When a continuation is captured, the stack cache is split by allocating a small data structure representing the captured continuation. The data structure points to the current continuation frame within the stack cache. The unused portion of the stack cache becomes the new stack cache, and an underflow frame is installed at its base. A throw is handled as in the incremental stack/heap strategy: the current stack cache is cleared, and some number of continuation frames are copied into it. The underflow frame at the base of the stack cache is linked to the portion of the new continuation that was not copied. This is a zero-overhead strategy, in which mutable variables generally cannot be allocated within a continuation frame, but continuation frames may be reused for multiple non-tail calls [19].

First-class continuations on the JVM

The implementations described in the previous section require to directly manipulate the stack, thus they are not suitable for being used on the Java Virtual Machine, which do not permit direct access or modification of stack contents. This section describes some implementation designed to implement first class continuations on the Java Virtual Machine.

Heap based model

In a typical implementation of a programming language, a true stack is used to record call frames. Each call frame consists at least of a return address, variable bindings and a link to the previous frame. The variable bindings are the actual parameters and local variables used by the called procedure. A call frame is typically built by the calling procedure (caller). The caller pushes on the stack the actual parameters, a link to its stack frame and the return address, then jumps to the called procedure (callee). The callee augments the frame by pushing values of local variables. If the callee in turn calls another routine, it creates a new stack frame in the same way. When the callee has reached the end of its code, it returns to the caller by resetting the frame link, removing the frame, and jumping to the saved return address. The state of each active call is recorded on the stack, and it is destroyed once the call has been completed [10].

Because of restricted access of stack content on the JVM, for languages that support first-class continuations this structure is not sufficient. First-class continuations require heap allocation of the call frames as well as the environment. This is

because the natural implementation of a continuation is to retain a pointer into the call stack. Because the continuation is a first-class object, there is no restriction on when it may be invoked. In particular, it may be invoked even after control has returned from the point where it was obtained. If so, the stack may have since grown, overwriting some of the stack frames in the continuation. The natural solution, then, is to maintain a linked list of heap-allocated stack frames. As the stack grows, a new frame is allocated in an unused portion of the heap so that the old stack frames remain intact [10].

The main disadvantage of heap allocation of call frames and environments is the overhead associated with the use of a heap. This overhead includes the additional cost of finding space in the heap when building the call frames and environments, the cost of storage reclamation to deallocate those frames and environments and the cost of following links instead of indexing a stack or frame pointer. The overhead also includes the indirect cost of using excessive amounts of memory. Furthermore, use of the heap rather than a stack prevents the use of some hardware-optimised or microcode-supported instructions for managing the stack [10].

The heap-based model has been used by several implementations, including Smalltalk, StacklessPython, Ruby, SML. On the JVM, this technique has been utilised by SISC [20], a fully R5RS compliant interpreter of Scheme, with proper tail-recursion and first-class continuations.

Continuations from continuation passing-style transform

An other approach to implement first-class continuations is to transform programs into continuation passing-style (CPS) [21, 22]. The standard CPS-transform is a whole-program transformation, in which all explicit or implicit return statements are replaced by function calls and all state is kept in closures. One effect of CPS is that the stack is completely bypassed during execution, and this is not ideal for a stack-based architecture like the JVM.

Considering that manually written CPS code shows that only a small number of functions in a program actually need to pass along continuations, Tiark Rompf et al. developed a selective CPS transform for the Scala programming language [23] that is applied only where it is actually needed, and allows to maintain a stack-based runtime discipline for the majority of code. Thus, they made use of Scala's pluggable typing facilities and introduce a type annotation, so that the CPS transform could be carried out by the compiler on the basis of expression types (i.e. it is type-directed). An advantage of this technique is that by design it avoids the performance problems associated with implementations of delimited continuations in terms of undelimited ones. However, there are some drawbacks. Because of the global transformation performed by the continuations compiler plugin, there are some control constructs that can not be used when calling a CPS function. For instance, using return statements in a CPS function may cause type mismatch compiler errors, thus is better to avoid using them. The compiler plugin

does not handle `try` blocks, so it is not possible to catch exceptions within CPS code [24].

There are also some issues with looping constructs. Capturing delimited continuations inside a while loop turns the loop into a general recursive function. Therefore each invocation of `shift` within a looping construct allocates another stack frame, so after many iterations it is possible to run into a stack overflow. Moreover, some looping constructs can not be used with a `shift` inside them, because everything on the call path between a `shift` and its enclosing `reset` must be CPS-transformed. That means that a `shift` cannot be used into the regular foreach, map and filter methods, because they are not CPS-transformed [24].

Continuations from generalized stack inspection

In [25], Pettyjohn et al. show how to translate a program into a form that allows it to capture and restore its own stack without requiring stack manipulation primitives. They demonstrate that the native exception handling mechanism can be used to propagate captured control state down the stack. Their work is an extension of previous work by Sekiguchi et al. [26] and Tao [27]. Variants of this technique has been described in [28] for JavaScript, in [29] for a Scheme interpreter targeting the .NET CLR and in Kilim [30, 31], a message-passing framework for Java. The basic idea is to break up the code into fragments (as top level methods) where the last instruction of any fragment is a call to the next fragment in the chain. To achieve this result, they have specialised continuation objects that maintain the state needed for each fragment and an overridden `Invoke` method to invoke the corresponding fragment. Each fragment knows exactly which fragment to invoke next [30]. The transform differs from continuation passing-style in that the call/return stack continues to be the primary mechanism for representing continuations; a heap representation of the continuation is only constructed when necessary. This may result in better performance than CPS-conversion for those programs that make only occasional use of first-class continuations [32].

This transformation preserves the calling signature of a procedure, but it augments the behavior of the procedure when a continuation is to be captured [32]. We therefore introduce into each method an additional control path that extracts the dynamic state of the method and appends it to a data structure. To capture a continuation, we throw a special exception to return control to the method along the alternate control path. After appending the dynamic state, the method re-throws the exception. This causes the entire stack to be emptied and the corresponding chain of reified frames to be built. A handler installed at the base of the stack is the ultimate receiver of the exception and it creates a first-class continuation object in the heap using the chain of reified frames [29].

This implementation technique is substantially equivalent to the stack strategy described in the first section of this chapter [32]. Moreover, it can nearly be a zero-overhead technique, for platforms in which exception handlers are not expensive,

especially when no exception is thrown. This is the case for the Java Virtual Machine [33].

The process consists of six steps [29, 32]:

1. Assignment Conversion - Capturing and re-instating a continuation will cause variables to be unbound and rebound multiple times. Variable bindings that are part of a lexical closure must not be unshared when this occurs. To avoid problems with unsharing that may occur when the stack is reified, assignment conversion converts assigned variables into explicit heap-allocated boxes, thereby avoiding problems with duplication of values. This conversion is best explained by showing it in Scheme source code:

   ```
   (lambda (x) ... x ... (set! x value) ...)
       =>
   (lambda (x)
       (let ((y (make-cell x)))
           ... (contents y) ... (set-contents! y value) ...))
   ```

   where `(make-cell x)` returns a new `cell` containing the value `x`, `(contents cell)` returns the value in `cell`, and `(set-contents! cell val)` updates `cell` with the new value `val`. After assignment conversion, the values of variables can no longer be altered - all side-effects are to data structures. This greatly simplifies the code transformation, because values may now be freely substituted for variables without having to first check to see whether they are assigned [22].

2. ANF Conversion - The code is converted to *administrative normal form* (A-normal form or ANF). Converting the code into A-normal form [34] gives names to the temporary values and linearizes the control flow by replacing compound expressions with an equivalent sequence of primitive expressions and variable bindings. After ANF conversion, all procedure calls will either be the right-hand side of an assignment statement or a return statement. For instance, the following Scheme code shows the ANF transformation of a very simple expression:

   ```
   (f (g x) (h y))
       =>
   (let ((v0 (g x)))
       (let ((v1 (h y)))
           (f v0 v1)))
   ```

   The following snippet shows the transformation for a fibonacci function in Java, considering as primitive subexpressions that can be evaluated without a method call:

```
int fib (int x) {
    if (x < 2)
        return x;
    else
        return fib (x - 2) + fib (x - 1);
}
    =>
int fib_an (int x) {
    if (x < 2)
        return x;
    else {
        int temp0 = fib_an (x - 2);
        int temp1 = fib_an (x - 1);
        return temp0 + temp1;
    }
}
```

3. Live variable analysis - We need to identify what variables are live at each continuation, i.e. at each fragment call. We are only interested in those variables that are live after a procedure or method call returns. For instance, in the previous code snippet, just before the last statement, `temp0` and `temp1` are alive, because they are used to compute the result to be returned. Conversely, `x` is no more live (is dead) as it is not used in the last statement. Unused or dead variables are not copied when the continuation is captured.

4. Procedure Fragmentation - For each actual procedure, we create a number of procedures each of which has the effect of continuing in the middle of the original procedure. This allows to restart execution right after each call site. Each procedure fragment will make a tail-recursive call to the next fragment. Fragmentation also replaces iteration constructs with procedure calls.

```
int fib_an (int x) {
    if (x < 2)
        return x;
    else {
        int temp0 = fib_an (x - 2);
        return fib_an0 (temp0, x);
    }
}

int fib_an0 (int temp0, int x) {
    int temp1 = fib_an (x - 1);
    return fib_an1 (temp1, temp0);
}

int fib_an1 (int temp1, int temp0) {
```

```
    return temp0 + temp1;
}
```

5. Closure conversion - A continuation is composed of a series of frames, that are closed over the live variables in the original procedure. Each frame also has a method that accepts a single value (the argument to the continuation) and invokes the appropriate procedure fragment. These closures can be automatically generated if the underlying language were to support anonymous methods.

```
abstract class Frame {

    abstract Object invoke(Object arg)
        throws Throwable;
}

class fib_frame0 extends Frame {

    int x;

    fib_frame0(int x) {
        this.x = x;
    }

    @Override
    Object invoke(Object return_value)
        throws ContinuationException, Throwable {
        return fib_an0(x);
    }

}
```

6. Code annotation - The fragmented code is annotated so that it can save its state in the appropriate continuation frame. Each procedure call is surrounded by an exception handler. This intercepts the special exception thrown for reifying the stack, constructs the closure object from the live variables, appends it to the list of frames contained by the special exception, and re-throws the exception. The calls in tail position are not annotated.

```
int fib_an (int x) {
    if (x < 2)
        return x;
    else {
        int temp0;
        try {
            temp0 = fib_an (x - 2);
```

```
        } catch (ContinuationException sce) {
            sce.extend (new fib_frame0 (x));
            throw sce;
        }
        return fib_an0 (temp0, x);
    }
}

int fib_an0 (int temp0, int x) {
    int temp1;
    try {
        temp1 = fib_an (x - 1);
    } catch (ContinuationException sce) {
        sce.extend (new fib_frame1 (temp0));
        throw sce;
    }
    return fib_an1 (temp1, temp0);
}

int fib_an1 (int temp1, int temp0) {
    return temp0 + temp1;
}
```

Java frameworks implementing continuations

Kilim

The Kilim framework [30, 31] provides lightweight actors, a type system that guarantees memory isolation between threads and a library with I/O support and synchronisation constructs and schedulers. It uses a restricted form of continuations that always transfers control to its caller but maintain an independent stack. Kilim implements a variant of generalized stack inspection [25]. It transforms compiled programs at the bytecode-level, inserting copy and restore instructions to save the stack contents into a separate data structure (called a *fiber*) when a continuation is to be accessed. Its implementation is based on threee main architectural choices:

Suspend-Resume

Kilim preserves the standard call stack, but provides a way to pause (suspend) the current stack and to store it in a continuation object called fiber. The fiber is resumed at some future time. Calling `Fiber.pause()` pops activation frames until it reaches the method that initiated `resume()`. This pair of calls is similar to the `shift` and `reset` operator from the literature on delimited continuations; they delimit the section of the stack to be saved.

Schedulable Continuations

Kilim actors are essentially thread-safe wrappers around Fibers. A scheduler chooses which Actor to resume on which kernel thread. Kernel threads are treated as virtual processors while actors are viewed as agents that can migrate between kernel threads.

Generators

Generators are essentially iterators that returns a stream of values. Each time we call a generator it gives us the next element. Kilim Generators are intended to be used by a single actor at a time, and run on the thread-stack of that actor. Even if the actor is running, it is prevented from executing any of its code until the generator yields the next element.

JavaFlow

The Apache Commons JavaFlow [35] is a library providing a continuations API for Java, accomplished via bytecode instrumentation which modifies the ordinary control flow of method calls to accomodate the ability to suspend and resume code execution at arbitrary points. JavaFlow transforms a method if it can reach a suspend() invocation. It transforms all non-pausable methods reachable from there as well, that are modified such that they can distinguish between normal execution, continuation capturing, and continuation resuming. This leads to inefficiencies, even when no continuations are used [36]. The instrumentation can be performed in advance or by a special class loader, which adds complexity either to the build process or to the application itself [30, 31].

RIFE

RIFE [37] is Java web application framework which allows web applications to benefit from first-class continuations. RIFE's pure Java continuation engine, which uses Java bytecode manipulation to implement continuations, has been extracted into a standalone Java library. It works similar to the Javaflow library, but it allows continuation capturing only within a specific method (`processElement`), so that there is always only one activation frame per continuation [36].

PicoThreads

A PicoThread is a lightweight, user-level Java thread that can be cooperatively-scheduled, dispatched and suspended [38]. PicoThreads are implemented in the Java bytecode language via a Java class-to-class translation. The translation produces threaded programs that yield control and a continuation sufficient to restart the thread where it left off. A PicoThread continuation is a Java object which contains a reference to the object and method in which it was created. Since Java's procedure

call stacks do not have dynamic extent, PicoThread continuations also contain extra state to store a method's local variables. PicoThread continuations extend Java exceptions, so that they can take advantage of Java's zero-cost exception mechanism to pass continuations from method to method. However the authors PicoThreads were unable to find a Java implementation fast enough to use the library effectively.

Matthias Mann's continuations library

Matthias Mann's continuations library implements continuations in Java using the ASM bytecode manipulation and analysis framework. The library provides an API allows to write coroutines and iterators in a sequential way [39].

Kawa's continuations

Kawa provides a restricted type of continuations, that are implemented using Java exceptions, and can be used for early exit, but not to implement coroutines or generators [40]. The following code, though different from the actual implementation, explains the concept:

```
class callcc extends Procedure1 {
    ...;
    public Object apply1(CallContext ctx) {
        Procedure proc = (Procedure) ctx.value1;
        Continuation cont
            = new Continuation (ctx);
        try {
            return proc.apply1(ctx);
            cont.invoked = true;
        } catch (CalledContinuation ex) {
            if (ex.continuation != cont)
                throw ex;  // Re-throw.
            return ex.value;
        }
    }
}
```

The `Procedure` that implements `call-with-current-continuation` creates a continuation object `cont`, that represents the current continuation, and passes it to the incoming Procedure `proc`. If `callcc` catches a `CalledContinuation` exception it means that `proc` invoked some `Continuation`. If it is the continuation of the current `callcc` instance, the code returns the value passed to the continuation; otherwise it re-throws the exception until a matching handler is reached.

The continuation is marked as `invoked`, to detect unsupported invocation of cont after `callcc` returns. (A complete implementation of continuations would instead copy the stack to the heap, so it can be accessed at a later time.)

```
class Continuation extends Procedure1 {
    ...;
    public Object apply1(CallContext ctx) {
        if (invoked)
            throw new GenericError
                ("Continuation can only be used once");
        throw new CalledContinuation (ctx.values, this, ctx);
    }
}
```

A `Continuation` is the actual continuation object that is passed to `callcc`; when it is invoked, it throws a `CalledContinuation` that contains the continuation and the value returned.

```
class CalledContinuation
    extends RuntimeException {
    ...;
    Object value;
    Continuation continuation;
    CallContext ctx;
    public CalledContinuation
        (Object value, Continuation cont, CallContext ctx) {
        this.value = value;
        this.continuation = cont;
        this.ctx = ctx;
    }
}
```

Chapter 3

Implementing first-class continuations on the JVM

"I don't care what anything was designed to do. I care about what it can do."

Apollo 13 (film, 1995)

The stack manipulation dilemma

The use of virtual machines for the implementation of programming languages has become common in recent compiler developments. Unlike low-level languages, such as C, that permit access to the stack through use of pointer arithmetic, higher level languages, such as Java or C# do not provide instructions for installing and saving the run-time stack. Compiling Scheme, or any other language that uses first-class continuations, to the JVM thus poses a challenging problem. At first glance, the implementers must either give up implementing continuations or manage a heap-stored stack. The former choice limits the programmers of these languages, besides automatically making the Scheme implementation non standard-compliant. The latter choice precludes many of the advantages that these machines supposedly offer. Indeed, the major problem with heap allocation of call frames and environments is the overhead associated with the use of a heap. This overhead includes the direct cost of allocating objects in the heap when building the call frames and environments, and of following references instead of increasing and decreasing a stack or frame pointer when accessing pieces of the frame or environment. The overhead also includes the indirect cost of garbage collection to manage stack frames and environments and the indirect cost of using significant amounts of memory. Furthermore, the use of the heap rather than a stack prevents the exploitation of commonly available hardware or microcode-supported stack push, pop and index instructions and the use of function call and return instructions.

A solution: generalises stack inspection

The idea is to fragment the original program in a sequence of atomic computations, then to throw an exception to unwind the stack, and use the same exception to store the list of computations that constitute the stack portion to be captured. This list of computations can be later used to restate the entire continuation. As an example, consider the stack of nested function calls in Figure 3.1. At top level we call `topLevel`, that in turn calls `f`, which calls `g`, which call `call/cc` with `h` as argument. `h` is a function that takes one argument. Each call is enclosed in an exception handler that catches a `ContinuationException`.

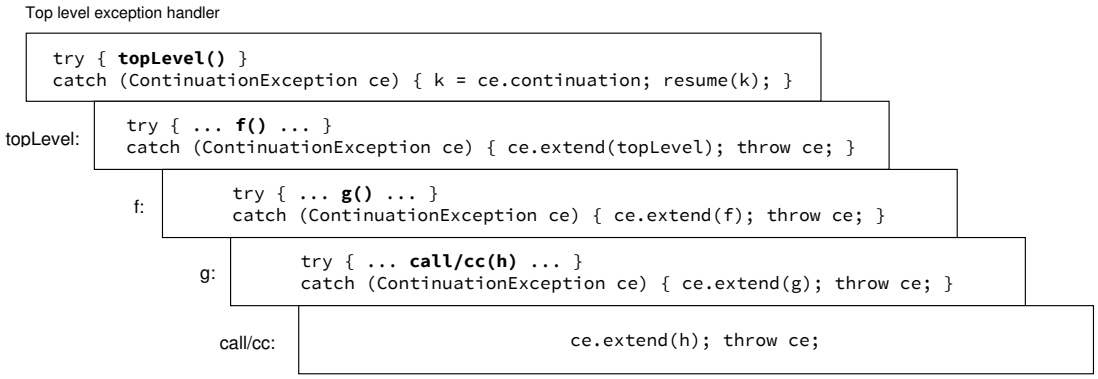


Figure 3.1

When the `call/cc` is called, it creates a new `ContinuationException` object, adds the computation associated to `h` to the list, and throws the exception. Just after the `throw`, the execution stops and the JVM searches routines in the stack for an exception handler. The first `try/catch` expression found, that is in `g`, extends the list with an other computation and re-throws the exception.

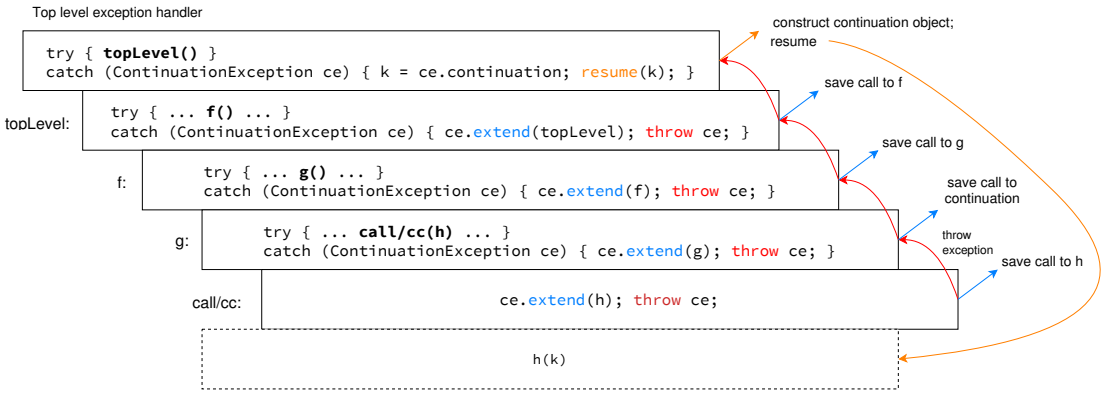


Figure 3.2

The exception goes past the try block to try blocks in an outer scope. At each step a new computation is added to the `ContinuationException`, until the control goes

to the top level exception handler, which assembles the actual exception object. Searching in outer scopes for exception handlers is called a *stack walk*. While the stack unwinds, the JVM pops the stack frames off of the stack, destroying all the stack allocated variables. However, as all the computation steps are saved in the `ContinuationException`, a copy of the stack is progressively created on the heap. The exception always maintains a reference to the list of computations during the stack walk, so that the continuation is not garbage-collected.

The top level handler, besides assembling the continuation object, resumes the execution of `h`, the function passed to the `call/cc`, passing to it the continuation as argument. If `h` does not invoke the continuation, the top level handler resumes the continuation after `h` returns. Figure 3.2 illustrates the process.

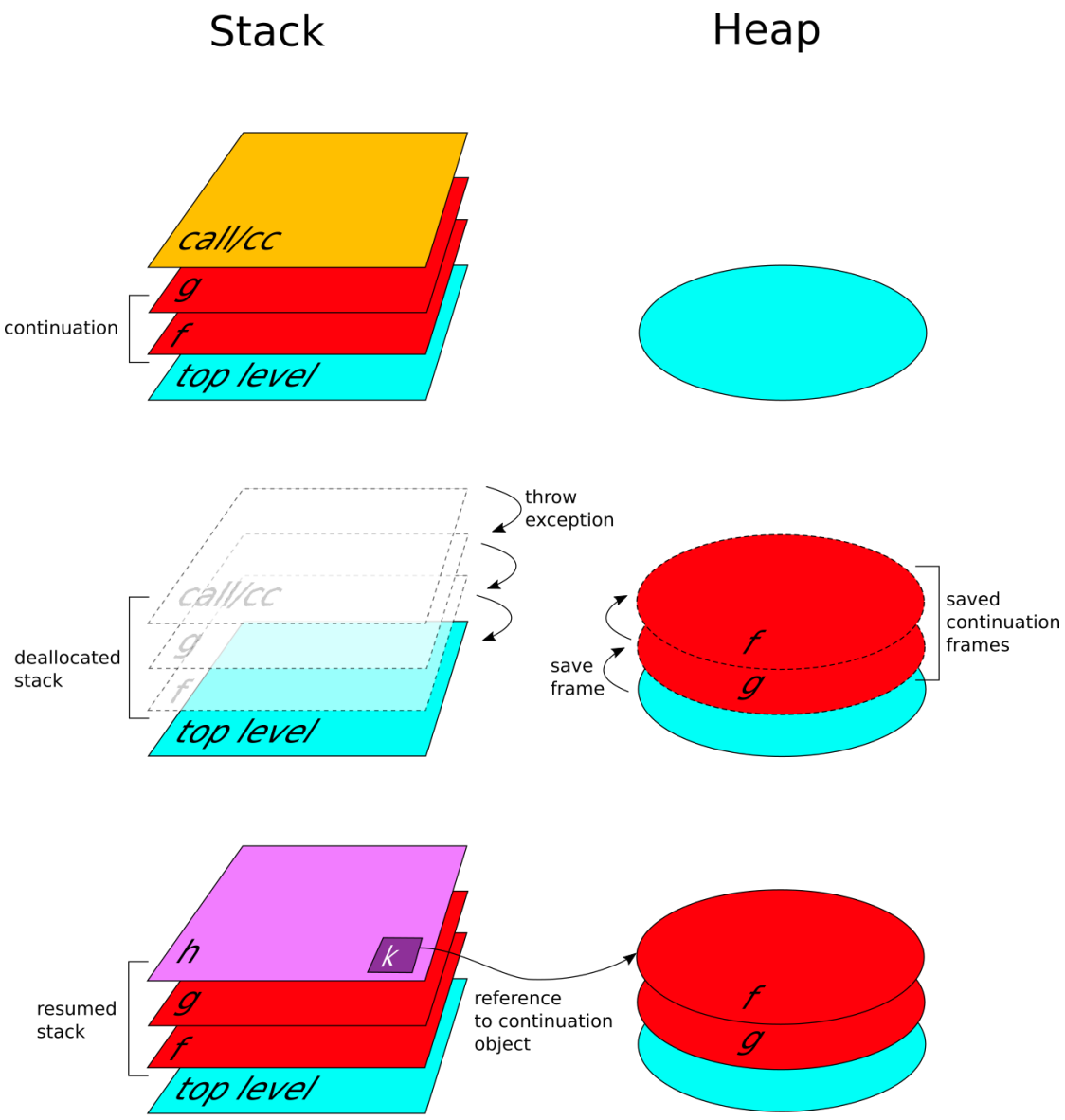


Figure 3.3: Stack and heap during a continuation capture

Figure 3.3 shows what happens in the stack and in the heap when a continuation is captured by `call/cc`. When `call/cc` is called the stack frames belonging to the continuation are under the `call/cc`'s one (assuming the stack growing bottom-up). Throwing the `ContinuationException`, `call/cc` starts to unwind the stack, and consequently the heap starts to be populated by the continuation frames. When top level is reached, the handler creates the continuation object. At the end of the process, the `h` function is resumed with the continuation object bound to its single argument.

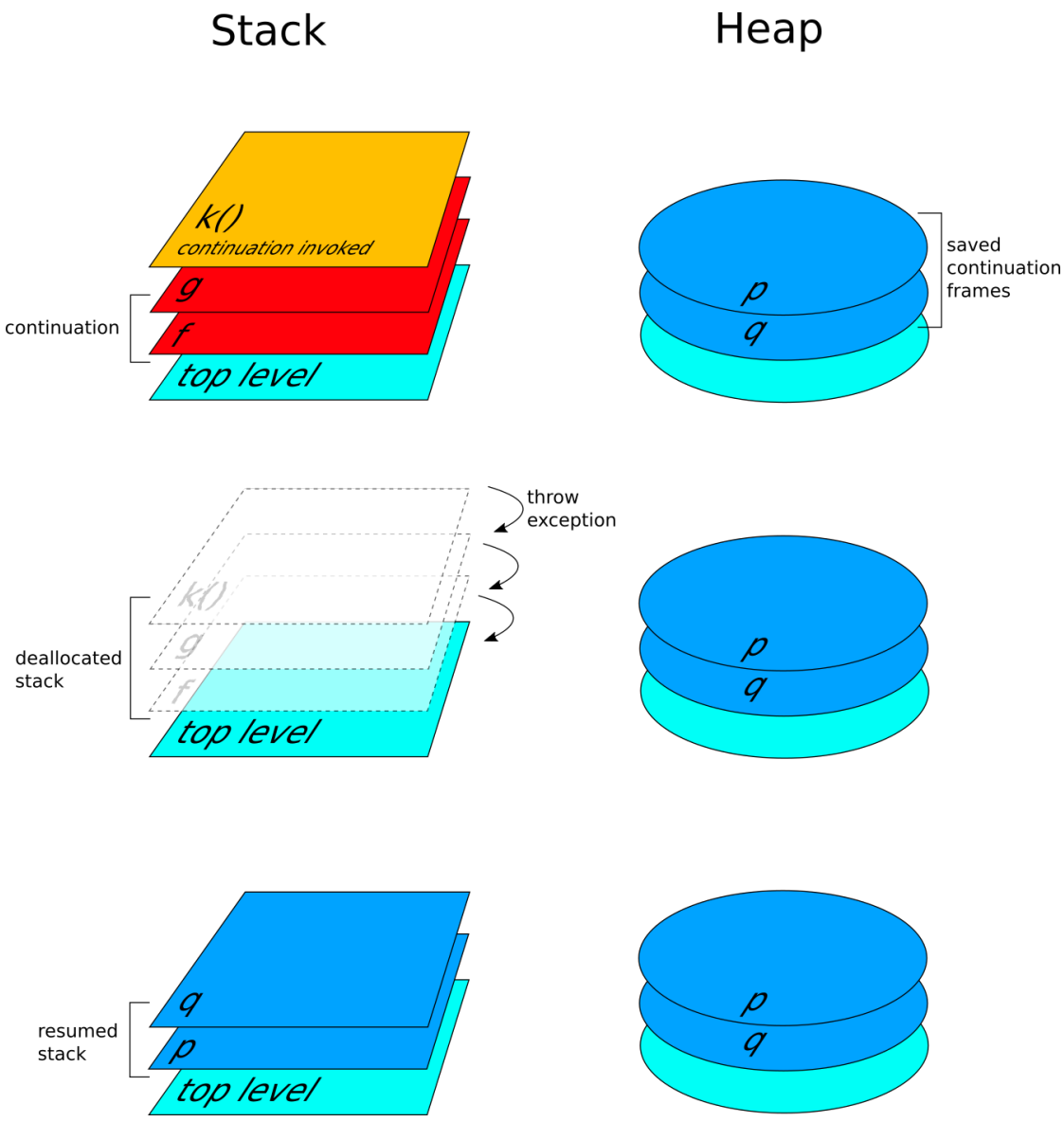


Figure 3.4: Stack and heap when reinstating a continuation

An interesting property of first-class continuations is that they can be invoked at any time, provided that they are saved in an accessible variable. When a continuation is invoked, it throws an `ExitException`. This causes the stack to be unwind, as in the capture case. The top level handler in this case resumes the continuation

frames stored in the continuation object. The final result is that the execution restart where it was suspended by the call/cc, while the heap continues to store the continuation object. This can be accessed other times, or can be garbage-collected if it is no more used.

Generalised stack inspection for a JVM-based Scheme

This section shows how the generalised stack inspection technique described by Pettyjohn et al. can be adapted to be used on the JVM, and how it can be included in a Scheme compiler. We will see also how some issues leaved open by the original paper have been tackled.

Assignment conversion

In our case, this step is not necessary. Indeed, management of shared variable bindings is an orthogonal issue with respect to our global transformation, and it is shared between all the languages that provide lexical closures. Kawa already supports lexical closures, so has its way of managing variable bindings. For each closure, Kawa creates a new class to represent a function together with the environment of captured variables.

A-Normalization

The first step of the process is to transform the source to A-normal form. ANF was introduced by Flanagan et al. in [34] as an intermediate representation for compilers. It encodes data flow explicitly by naming all sub-expressions within the program and permitting only a single definition of any particular variable. The paper by Flanagan et al. also presents a basic linear-time A-normalization algorithm for a subset of Scheme. The algorithm can be easily extended to handle top-level defines and side effects [41]. Being Kawa a super-set of R7RS Scheme and having also many Java related extensions, the code of the original A-normalizer must be further extended. Instead of performing the transformation directly on the Scheme source, I opted for performing it on the abstract syntax tree, as it already uses a reduced set of expression types.

The following code shows an instance of the transformation in three steps. The *return* operation corresponds to the identity function, while the *bind* operation is a function that constructs let bindings to make every atomic computation explicit (the bind here has the same purpose as the `normalizeName` function in the Flanagan et al. paper).

The syntax tree is traversed starting from the root, and each non-atomic expression is passed to the `bind` with another parameter, called *context* (the very first context

is the identity function that returns its argument). The context is a function that can be invoked. When the visiting process reaches a non-atomic expression a new context is created, and the passed context is called only inside the new one. The `bind` function has two purposes:

1. to create a context, that generates a `let` expression to let-bind the expression next to come in the traversing process;
2. to visit the passed expression, to continue the syntax tree traversing.

This chain finish when a leaf (an atomic expression) is encountered in the tree, in this case the passed context is invoked (which in turn will invoke the previous context and so on). At this point the chain of context invocations starts to wrap each expression in a `let` binding, processing the expressions backward, and enclosing them step by step in nested `let` expressions. This backward traversing stops when the context called is the identity function. This happens in the leaves.

When the expression to normalize is a conditional, the `bind` is used on each branch expression. Instead of creating a let binding for each branch, as they cannot be evaluated before the test outcome, `bind` calls the visit method with the identity context, restarting the normalization in each branch.

The following code shows the a-normalization process for a simple Scheme expression. Note that the internal `if` expression should be further normalized, but we consider it atomic here, to keep the example short. The algoritm performs a monadic transformation combining three steps:

1. Monadic conversion:

```
(+ 1                                  (bind (if (>= x 0)
   (if (>= x 0)                             (f x)
       (f x)              -->               (return 0))
       0))                                (lambda (t) (+ 1 t)))
```

2. The result is interpreted in the identity monad:

```
             (return a)  =>  a

(bind a (lambda (x) b))  =>  (let ((x a)) b)


(bind (if (>= x 0)                    (let ((t (if (>= x 0)
          (f x)          -->                    (f x)
          (return 0))                           (return 0))))
       (lambda (t) (+ 1 t)))             (+ 1 t))
```

3. Nested let are flattened:

```
(let ((x (let ((y a))              (let ((y a))
           b)))          -->       (let ((x b))
  c)                                 c))
```

Code fragmentation

This transformation, working on code previously A-normalized, fragments the code in a sequence of function calls. Each let-bind expression is enclosed in a lambda closure that accepts one argument. The argument is an other lambda closure that has in the body the call to the next code fragment. In this way the original source is rewritten as a sequence of function calls, each call representing a computation step. This way of fragmenting the source allows to avoid defining many top level procedures, that would also require an additional pass to perform live variable analysis.

An example of the entire transformation is showed below:

1. original source

```
(define incr #f)

(+ (call/cc
      (lambda (k)
          (set! incr k)
          0))
  1) ; => 1
```

2. after A-normalization

```
(let ((v1 (lambda (k)              ; computation #1
        (let ((v0 (set! incr k)))
          0))))
 (let ((v2 (call/cc v1)))          ; computation #2
   (+ v2 1))))                     ; computation #3
```

3. after fragmentation

```
((lambda (incr_an1)                ; fragment #1
  (let ((v1 (lambda (k)
              (let ((v0 (set! incr k)))
                0))))
     (incr_an1 v1)))
 (lambda (v1)
   ((lambda (incr_an2)             ; fragment #2
```

```
    (let ((v2 (call/cc v1)))
      (incr_an2 v2)))
  (lambda (v2)
    (+ v2 1)))))                    ; fragment #3
```

Live variable analysis and closure conversion

Kawa's support for lexical closures allows to completely avoid these steps. Each fragment, created as described in the previous section, is closed over the values of the variables that are live at that point.

A continuation will be composed of a series of frames. A *frame* is an object with a method that accepts a single value (the argument to the continuation) and invokes the appropriate procedure fragment, to continue the computation from the capture point. Also these frames will be closed over the next fragment to call.

Code Instrumentation

Beside fragmentation, instrumentation is performed for installing exception handlers around each computation step to enable the capture and resume of continuations. Kawa supports `try-catch` expressions, which are translated directly to native `try/catch` statements in Java bytecode. A `try-catch` expression is created around each computation to capture a possible `ContinuationException`. The installed exception handler adds a new frame (an invokable object enclosing a call the next computation step) to the list of frames included inside the `ContinuationException` object, then rethrows the exception.

The following code resembles the final result after instrumentation:

```
((lambda (incr_an1)
  (let ((v1 (lambda (k)
              (let ((v0 (set! incr k)))
                0))))
    (incr_an1 v1)))
 (lambda (v1)
   ((lambda (incr_an2)
      (let ((v2 (try-catch (call/cc v1)             ; try/catch
                  (cex <ContinuationException>      ; handler
                    (let ((f (lambda (continue-value)
                               (incr_an2 continue-value))))
                      (cex:extend (<ContinuationFrame> f))
                      (throw cex))))))              ; re-throw
        (incr_an2 v2)))
    (lambda (v2)
      (+ v2 1)))))
```

Issues

`call/cc` in higher order functions

Since Kawa optimise some built-in procedures (like `map`, `foreach` and `filter`) implementing them as Java methods, and because of the global transformation needed by the `call/cc`, continuations cannot be captured inside functions passed to those higher order functions. Indeed, the Java implementation of map (`gnu.kawa.functions.Map`) is not 'aware' of continuations, thus when you use `call/cc` inside the lambda passed to `map`, it will not be able to handle a `ContinuationException`, resulting in a runtime error.

In the next chapter, we will see a possible solution to this problem.

Code size

The creation of fragments will introduce a number of extra code. Although the overhead should be small, there will be an increase in code size proportional to the number of code fragments.

Code instrumentation introduces a number of `try/catch` blocks. This will also increase code size proportional to the number of code fragments. Chapter 6 will present an estimate of the code size increase.

Integration

Given that the code transformation needed to support continuations adds a overhead to the compilation process and the generated bytecode, it is necessary to implement A-normalization and instrumentation as optional passes, that can be enabled only when we want to use `call/cc`. This adds the challenge of integrating such a global transformation in Kawa, avoiding to make too many changes to the compiler.

Chapter 4

A call/cc implementation for Kawa

"Do... or do not. There is no try."

The Empire Strikes Back (film, 1980)

An instance of the transformation in Java

As a first preliminary step, I ported the C# code in [32] to Java, to study the feasibility of the technique on the JVM. The code represents a single instance of the transformation for a simple fibonacci function, and implements some support functions and data structures. Given that the global transformation fragments the original source in many function calls, I produced four versions of the transformed code, to compare the performance of different type of calls on the JVM:

1. The first one uses nested static classes to implement the continuation frames of the function to be run:

```
class fib_frame0 extends Frame {

    int x;

    public fib_frame0(int x) { this.x = x; }
    @Override
    public Object invoke(Object return_value)
            throws ContinuationException, Throwable {
        // call to the next fragment
        return fib_an0(x);
    }
}
```

```
public int fib_an(int x)
      throws ContinuationException, Throwable {
    try {
        pause();
    } catch (ContinuationException sce) {
        sce.extend(new ContinuationFrame(new fib_frame0(x)));
        throw sce;
    }

    return fib_an0(x);
}
```

2. the second version uses `MethodHandle`s, that were introduced in Java 7. A `MethodHandle` is a typed, directly executable reference to an underlying method, constructor or field:

```
static Object fib_frame0_invoke(Object x, Object continue_value)
        throws SaveContinuationException, Exception {
     return fib_an0 ((int) x);
}

static MethodHandle fib_frame0(int x)
        throws Exception {
    MethodType mt = MethodType.methodType(Object.class,
                                          Object.class,
                                          Object.class);
    MethodHandle handle = lookup.findStatic(fib_mh.class,
                                            "fib_frame0_invoke",
                                            mt);

    return handle.bindTo(x);
}

public static int fib_an(int x)
       throws SaveContinuationException, Exception {
    try {
        pause();
    } catch (SaveContinuationException sce) {
        sce.Extend(new ContinuationFrame(fib_frame0(x)));
        throw sce;
    }

    return fib_an0(x);
}
```

3. the third version uses Java 8 lambdas, specified with the new Java syntax:

```
static Object fib_frame0_invoke(Object x, Object continue_value)
        throws SaveContinuationException, Exception {
     return fib_an0 ((int) x);
}

static Frame fib_frame0(int x)
        throws Exception {
    Frame f = (Object continue_value)
                -> {
                     return fib_frame0_invoke(x, continue_value);
                   };
    return f;
}

public static int fib_an(int x)
        throws SaveContinuationException, Exception {
    try {
        pause();
    } catch (SaveContinuationException sce) {
        sce.Extend(new ContinuationFrame(fib_frame0(x)));
        throw sce;
    }

    return fib_an0(x);
}
```

4. the last version generates lambdas explicitly using LambdaMetafactory, an API introduced in Java 8 to facilitate the creation of simple function objects.

```
fib_frame0_factory
    = LambdaMetafactory
        .metafactory(lookup,
                     "invoke",
                     invokedType,
                     methodType,
                     lookup.findStatic(fib_meta.class,
                                       "fib_frame0_invoke",
                                       implType),
                     methodType).dynamicInvoker();

static Object fib_frame0_invoke(Object x, Object continue_value)
        throws SaveContinuationException, Throwable {
     return fib_an0 ((int) x);
```

```
}

static Frame fib_frame0(int x)
        throws Throwable {
    return (Frame) fib_frame0_factory.invoke(x);
}
```

I tested each type of method call with JMH [42, 43], a benchmarking framework for the JVM. Figures 4.1 , 4.2 show the results. The lambda case is quite fast, if compared with MethodHandles, but also the explicit use of LambdaMetafactory gives good results, provided that the call to LambdaMetafactory.metafactory is cached in a static field. However, the difference in performance between lambda calls and regular method calls is negligible. Thus is not worth to re-design a significant part of the compiler, and to loose the compatibility with previous version of the JVM, for such a small improvement.

Benchmark	Score (ns/op)	Error (ns/op)
Direct field access	3.13	0.02
Unsafe field access	3.14	0.03
Field direct access by reflection	5.34	0.04
Field getter by reflection MethodHandle call	8.71	0.32
Regular static method call	14.80	0.18
Lambda call	15.69	0.14
Metafactory generated lambda call	16.75	0.18
Regular method call	17.16	0.24
Reflection method (by getMethod) call	20.12	0.14
MethodHandle of static method call	20.60	0.17
MethodHandle	22.10	0.51

Figure 4.1: Performance comparison of different types of call in Java

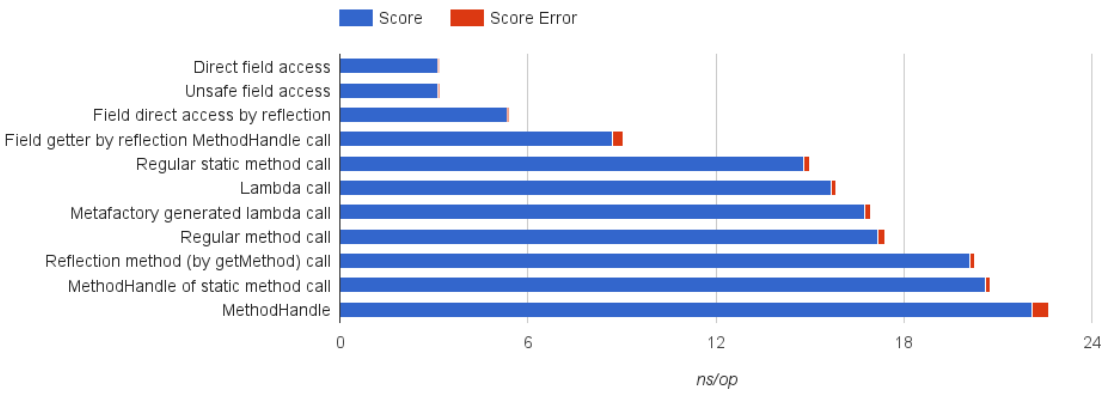


Figure 4.2: Performance comparison of different types of call in Java

Exceptions performance in Java

The capture of a continuation, and in particular the stack copying mechanism, is driven by exception throwing and exception handling. Therefore, is crucial to understand how the installation of exception handlers and the construction of an Exception object impact the performance.

In Java, when throwing an exception, the most expensive operation is the construction of the stack trace, that is useful for debugging reasons. As well as we are not using exceptions with they original purpose, we can have rid of the stack trace construction and optimise the `Exception` object. It is sufficient to override the `fillInStackTrace` method of `Throwable`:

```
public static class FastException extends Exception {

    @Override
    public Throwable fillInStackTrace() {
        return this;
    }
}
```

I performed a straightforward benchmark, comparing the time spent by a regular method call, a method call surrounded by an exception handler, a method call throwing a caught exception and a method call throwing a `FastException`.

```
    // case 1
    t.method1(i);

    // case 2
    try {
        t.method2(i);
```

```
} catch (Exception e) {
    // We will *never* get here
}

// case 3
try {
    t.method3(i);
} catch (Exception e) {
    // We will get here
}

// case 4
try {
    t.method4(i);
} catch (FastException e) {
    // We will get here
}
```

The results from 10 million iterations are shown in the following table.

	time (ms)
regular	1225
no caught exception	1240
caught exception	35482
caught, optimised	1330

As you can see, to catch a `FastException` introduces a negligible overhead, while instantiating an `Exception` with its stack trace is more than an order of magnitude more expensive.

Support code

For capturing and resuming continuations we need a framework to support all the required operations, such as construct an object that models the continuation, and turn a continuation object back into an actual continuation.

```
public static class ContinuationFrame {

    Procedure computation;
    ArrayList<ContinuationFrame> continuation;

    public ContinuationFrame(Procedure frame) {
        computation = frame;
```

```
        }
    }
```

The basic blocks of a continuation are its `ContinuationFrame`s. A `ContinuationFrame` (for brevity, a frame) is a simple data structure which contains a single computation (a `Procedure` that takes one argument), and a list of `ContinuationFrame`s. The list is used by the next capture of a continuation. All the frames needed to assemble a continuation are collected using a `ContinuationException`. This class extends `FastException` and stores the list of frames which is extended step by step by the chain of throws. It contains also a list of frames that have been already reloaded by a previously call to `call/cc`. When the exception reaches the top level exception handler, this calls the method `toContinuation` that builds a new `Continuation` object using the two lists.

```
public static class ContinuationException extends FastException {

    ArrayList<ContinuationFrame>
    newCapturedFrames = new ArrayList<ContinuationFrame>();

    ArrayList<ContinuationFrame> reloadedFrames;

    public void extend(ContinuationFrame extension) {
        newCapturedFrames.add(extension);
    }

    public void append(ArrayList<ContinuationFrame> oldFrames) {
        reloadedFrames = oldFrames;
    }

    public Continuation toContinuation() throws Exception {
        return new Continuation(newCapturedFrames,
                                reloadedFrames);
    }
}
```

The `Continuation` constructor takes the two lists and assembles the continuation.

```
public class Continuation extends Procedure0or1 {

    ArrayList<ContinuationFrame> frames;

    public Continuation(ArrayList<ContinuationFrame> newFrames,
                        ArrayList<ContinuationFrame> oldFrames) {

        frames = (oldFrames != null)
```

```
                ? new ArrayList<ContinuationFrame>(oldFrames)
                : new ArrayList<ContinuationFrame>();

        for(int i = newFrames.size()-1; i >= 0; i--) {
            ContinuationFrame newFrame = newFrames.get(i);
            if (newFrame.continuation != null) {
               throw new Error("Continuation should be empty here");
            }
            newFrame.continuation
               = new ArrayList<ContinuationFrame>(frames);
            frames.add(newFrame);
        }
    }
```

When a continuation is invoked, we actually call the `apply` method of `Continuation`. Here we create a new procedure which, when called, resumes the continuation. We wrap the procedure in an exception so that, throwing it, we unload the *current* continuation. The top level handler will receive this exception and will use it to resume the *invoked* continuation.

```
    public Object apply0() throws Throwable {
        return apply1(Values.empty);
    }

    public Object apply1(final Object val) throws Throwable {

        Procedure t = new Procedure1() {

            public Object apply1(Object ignored) throws Throwable {
                return reloadFrames(frames.size()-2, val);
            }
        };

        throw new ExitException(t);
    }
```

The `Continuation` object also contains the method to resume the continuation. `reloadFrames` iterates over the list of frames in reverse order to re-establish the saved continuation reconstructing the stack. The topmost frame gets the restart value passed into it.

```
    Object resume(final Object restartValue) throws Throwable {
        return reloadFrames(frames.size()-1, restartValue);
    }
```

```
    Object reloadFrames(int endIndex, Object restartValue)
    throws Throwable {
        Object continueValue = restartValue;
        for (int i = endIndex; i >= 0; i -= 1) {
            ContinuationFrame frame = frames.get(i);
            try {
                continueValue = frame.computation
                                        .apply1(continueValue);
            } catch (ContinuationException sce) {
                sce.append(frame.continuation);
                throw sce;
            }
        }
        return continueValue;
    }

}
```

`TopLevelHandler` deals with running top level calls in an exception handler that catches instances of `ContinuationException`, thrown by `call/cc`, and `ExitException`, thrown by a continuation invocation. In the first case it creates a continuation object and resumes the execution of the function passed to `call/cc`. In the second case it calls the function enclosed in the `ExitException`, which reinstates the continuation.

```
public class TopLevelHandler extends Procedure1 {

    public Object apply1(Object arg1) throws Throwable {
        return runInTopLevelHandler((Procedure) arg1);
    }

    public void compile(...) { ... }

    public static Object runInTopLevelHandler(Procedure initialFrame)
    throws Throwable {
        while (true) {
            try {
                return invokeFrame(initialFrame);
            } catch (ExitException rce) {
                initialFrame = rce.thunk;
            }
        }
    }

    private static Object invokeFrame(final Procedure initialFrame)
```

```
    throws Throwable {
        try {
            return initialFrame.apply1(null);
        } catch (ContinuationException sce) {
            final Continuation k = sce.toContinuation();

            Procedure f = new Procedure1() {

                public Object apply1(Object arg) throws Throwable {
                    return k.resume(k);
                }
            };

            throw new ExitException(f);
        }
    }
}
```

The `CallCC` procedure implements `call/cc`. It throws a new `ContinuationException`, saving in it the `call/cc` argument (a `Procedure` object).

```
public class CallCC extends Procedure1 {

    public Object apply1(Object arg1) throws Throwable {
        return call_cc((Procedure) arg1);
    }

    public void compile(...) { ... }

    public static Object call_cc(final Procedure receiver)
    throws ContinuationException {
        try {
            throw new ContinuationException();
        } catch (ContinuationException sce) {
            sce.extend(new ContinuationFrame(receiver));
            throw sce;
        }
    }
}
```

A significant variation with respect to the implementation proposed by Pettyjohn et al. is that the function that resumes the stack frames is implemented using iteration instead of recursion. This avoids using too much stack, as the JVM, differently from the C# MSIL, does not support tail call optimisation. Another difference is in the representation of the list of frames. Instead of using a linked

list adding elements at the beginning, I used a Java `ArrayList`, adding elements at the end of the list. This allows to avoid reversing a list at every capture, and saves an object allocation at each list extension.

A brief overview of Kawa's compilation process

In Kawa there are mainly five compilation stages [40]:

1. Syntactic analysis - the first compilation stage reads the source input. The result is one or more Scheme forms (S-expressions), represented as lists.

2. Semantic analysis - the main source form is rewritten into a set of nested `Expression` objects, which represents Kawa's *abstract syntax tree* (AST). For instance, a `QuoteExp` represents a literal, or a quoted form, a `ReferenceExp` is a reference to a named variable, an `ApplyExp` is an application of a procedure func to an argument list and a `LetExp` is used for let binding forms. The Scheme primitive syntax lambda is translated into a `LambdaExp`. Other subclasses of `Expression` are `IfExp`, used for conditional expressions, `BeginExp`, used for compound expressions and `SetExp`, used for assignments. The top-level `Expression` object is a `ModuleExp` and can be considered the root of the AST. This stage also handles macro expansion and lexical name binding.

3. Optimisation - an intermediate pass performs type-inference and various optimisation, such as constant folding, dead code elimination and function inlining.

4. Code generation - the `ModuleExp` object is translated into one or more byte-coded classes. This is done by invoking a `compile` method recursively on the `Expression`s, which generates JVM instructions using the bytecode package, writing out the resulting class files.

5. Loading - if the code is compiled and then immediately executed, the compiled code can be immediately turned into Java classes using the Java `ClassLoader` feature. Then the bytecode can be loaded into the Kawa run-time.

A-Normalization

I created a new `ExpVisitor` that manipulates the syntax tree implementing the transformation to ANF, already described in chapter 3. An `ExpVisitor` is Java class that can be extended to implement code that traverses the AST to apply a certain transformation. The new visitor, called `ANormalize`, performs the A-normalization pass just before the optimisation stage of the compiler.

```
[... parsing ...]

ANormalize.aNormalize(mexp, this); // <-- A-normalization
InlineCalls.inlineCalls(mexp, this);
ChainLambdas.chainLambdas(mexp, this);
FindTailCalls.findTailCalls(mexp, this);
```

```
[... code generation  ...]
```

At first, we call the visit function on root of the AST, passing as context the identity function.

```
public static void aNormalize(Expression exp, Compilation comp) {
    [...]
    visitor.visit(exp, identity);
}
```

The core of the A-normalizer is the `bind` function, already introduced in Chapter 3, here called `normalizeName`. `normalizeName` creates a new context, then it will visit the expression with this new context. If the passed expression is atomic (cannot be further normalized), like a literal or an identifier, the new context calls the old context with the expression as input. Otherwise it creates a new `let` expression, binds the expression to a new variable in the `let` (with `genLetDeclaration`), then replaces every occurrence of the expression in the code with a reference to the just created variable (with `context.invoke(new ReferenceExp(decl))`).

```
protected Expression normalizeName(Expression exp,
                                   final Context context) {
    Context newContext = new Context() {
        @Override
        Expression invoke(Expression expr) {
            if (isAtomic(expr))
                return context.invoke(expr);
            else {
                // create a new Let
                LetExp newlet = new LetExp();

                // create a new declaration in the let, using
                // the new expression value
                Declaration decl = genLetDeclaration(expr, newlet);

                // occurrences of expr in the next computation are
                // referenced using the new declaration
                newlet.body = context.invoke(new ReferenceExp(decl));
                return newlet;
            }
        }
    };

    return visit(exp, newContext);
}
```

When the expression to normalize is a conditional, as its branches cannot be evaluated before the test outcome, we use `normalizeName` on each branch expression. Instead of creating a new variable for each branch, we restart the normalization in each branch.

```
protected Expression visitIfExp(final IfExp exp,
                                final Context context) {
    Context newContext = new Context() {

        @Override
        Expression invoke(Expression expr) {
            exp.then_clause = normalizeTerm(exp.then_clause);
            exp.else_clause = (exp.else_clause != null)
                              ? normalizeTerm(exp.else_clause)
                              : null;

            exp.test = expr;

            return context.invoke(exp);
        }
    };
    return normalizeName(exp.test, newContext);
}
```

When an atomic expression is encountered in the tree, the passed context is directly invoked with expression passed as argument. At this point the chain of context invocations starts to wrap each expression in a let binding, traversing the AST backward, nesting each non atomic expression in a new `let`.

```
protected Expression visitQuoteExp(QuoteExp exp,
                                   Context context) {
    return context.invoke(exp);
}

protected Expression visitReferenceExp(ReferenceExp exp,
                                       Context context) {
    return context.invoke(exp);
}
```

Code fragmentation

Another `ExpVisitor`, `FragmentAndInstrument`, performs the fragmentation and the instrumentation. As described in Chapter 3, the new visitor transforms the code in a sequence function calls. At the same time it wraps in a `try-catch`

expression every atomic computation it encounters in the traversing. This stage is inserted between the A-normalization and the optimisation pass.

```
[... parsing ...]

ANormalize.aNormalize(mexp, this);
FragmentAndInstrument.fragmentCode(mexp, this);// <-- fragmentation
                                                //and instrumentation
InlineCalls.inlineCalls(mexp, this);
ChainLambdas.chainLambdas(mexp, this);
FindTailCalls.findTailCalls(mexp, this);

[... code generation ...]
```

The transformation starts at the root of the AST (a `ModuleExp`), and continues analysing each node of the tree recursively. The most relevant method in `FragmentAndInstrument` is `visitLetExp`, which deals with the transformation of `let` expressions.

```
protected Expression visitLetExp(LetExp exp, Void ignored) {
    Declaration letDecl = exp.firstDecl();
    Expression nextExp = exp.body;
    Expression continueValue = letDecl.getInitValue();
```

After A-normalization the code is mainly made by nested `let` expression that bind to a variable every atomic computation. `visitLetExp` takes a `LetExp` and transforms it in two closures, applying the first to the second one. The former closure executes an atomic computation and calls the latter closure. The latter closure contains the original body of the `let` expression, which will be further fragmented. Using the example from Chapter 3:

```
((lambda (incr_an1)        ;   <--   closure #1
  (let ((v1 (lambda (k)
              (let ((v0 (set! incr k)))
                0))))
     (incr_an1 v1)))
 (lambda (v1)              ;  <--   closure #2
   ((lambda (incr_an2)
      (let ((v2 (call/cc v1)))
        (incr_an2 v2)))
    (lambda (v2)
      (+ v2 1)))))
```

The following code creates the first closure. It simply generates a new lambda expression that takes an argument. The original `LetExp` becomes the body of the lambda.

```
Declaration nextFragmentDecl = new Declaration("continue-fragment");
LambdaExp fragment = new LambdaExp(1);
fragment.body = exp;
fragment.addDeclaration(nextFragmentDecl);
```

We replace the let body with the call to the next fragment, that is `(incr_an1 v1)` in the previous Scheme example.

```
exp.body = new ApplyExp(applyRef,
                        new ReferenceExp(nextFragmentDecl),
                        new ReferenceExp(letDecl));
```

The code that creates the second closure is very similar to which that generates the first one. It is another new lambda expression that takes an argument. The body this time is the body of the original `LetExp`.

```
Declaration continueValueDecl = new Declaration("continue-value");
LambdaExp nextFragment = new LambdaExp(1);
nextFragment.body = nextExp;
nextFragment.addDeclaration(continueValueDecl);
```

We create a new function call, which applies the first lambda to the second one.

```
    ((lambda (incr_an1)        ;   <--   closure #1
       ...)
     (lambda (v1)              ;  <--   closure #2
       ...))

ApplyExp fragmentCall = new ApplyExp(fragment,
                                     nextFragment);
```

Then we can move one to annotate with a `try-catch` the `let` binding (see next section), and to traverse the rest of the tree calling visit on the body of the second lambda.

```
Expression annotatedExp = visitAndAnnotate(continueValue,
                                           nextFragmentDecl);
letDecl.setInitValue(annotatedExp);

// visit the rest of the code.
nextFragment.body = visit(nextFragment.body, ignored);

return fragmentCall;
```

Code Instrumentation

First of all, each top level expression is wrapped inside a `TopLevelHandler` call, which surrounds the expression with an exception handler, as seen in Chapter 3.

```
protected Expression visitModuleExp(ModuleExp exp, Void ignored) {

    if (exp.body instanceof ApplyExp
        && ((ApplyExp)exp.body).isAppendValues()) {
        ApplyExp body = ((ApplyExp)exp.body);
        for (int i = 0; i < body.args.length; i++) {
            body.args[i] = installTopLevelHandler(visit(body.args[i],
                                                        ignored));
        }
        return exp;
    }

    exp.body = installTopLevelHandler(visit(exp.body, ignored));

    return exp;
}
```

Then we perform the main part of instrumentation in the `visitAndAnnotate` method, which we call on every `let` binding, as shown in the previous section. In `visitAndAnnotate`, we create a `TryExp` and an exception handler that catches `ContinuationExceptions`.

```
private Expression visitAndAnnotate(Expression exp,
                                    Declaration nextFragmentDecl) {
    TryExp annotatedExp = new TryExp(exp, null);
    Declaration handlerDecl = new Declaration((Object) null,
                                              contExpceptionType);
    ReferenceExp handlerDeclRef = new ReferenceExp(handlerDecl);
```

We also create the frame needed to extend the `ContinuationException`. The frame computation is a lambda which contains the call to the next fragment. Then we can generate the code to create a `ContinuationFrame` with the lambda just created. The lambda will be translated to a `Procedure` object at runtime.

```
(try-catch (call/cc v1)              ; try/catch
  (cex <ContinuationException>       ; handler
    (let ((f (lambda (continue-value)
               (incr_an2 continue-value))))
      (cex:extend (<ContinuationFrame> f))
      (throw cex))))                 ; re-throw
```

```
Declaration argDecl = new Declaration("continue-value");
ApplyExp nextFragmentCall = new ApplyExp
                                  (new ReferenceExp(nextFragmentDecl),
                                   new ReferenceExp(argDecl));

Expression frame = createFrame(argDecl, nextFragmentCall);

ApplyExp cframe = new ApplyExp(contFrameClass,
                               frame);
ApplyExp extend = new ApplyExp(new PrimProcedure("Helpers",
                                                 "extend", 2),
                               handlerDeclRef,
                               cframe);
```

The last thing to generate is the re-throw instruction for the caught `ContinuationException`. Eventually, we visit the annotated exp to continue the tree traversing.

```
ApplyExp throwApply = new ApplyExp(primitiveThrow,
                                   handlerDeclRef);
Expression begin = new BeginExp(extend, throwApply);
annotatedExp.addCatchClause(handlerDecl, begin);

// visit the wrapped expression
annotatedExp.try_clause = visit(annotatedExp.try_clause, null);
return annotatedExp;
```

Other control operators: delimited continuations

The transformation and the support code described in this Chapter is not only suitable to implement `call/cc`, but it can also be employed to implement other control operators.

Prompts and barriers

A *prompt* is a special kind of continuation frame that is annotated with a specific tag. Some operations allow to save continuation frames from the capture position out to the nearest enclosing prompt; such a continuation is sometimes called a delimited continuation [44].

A *continuation barrier* is another kind of continuation frame that prohibits certain replacements of the current continuation with another. A continuation can be replaced by another only when the replacement does not introduce any continuation barriers. A continuation barrier thus prevents to jump into a continuation that is protected by a barrier [44].

call-with-continuation-prompt

I implemented a simple version of the `call-with-continuation-prompt` procedure. This function installs a prompt, and then it evaluates a given thunk under the prompt. During the dynamic extent of the call to thunk, if a user calls `call/cc`, the stack will be unwind until the prompt. Thus `call/cc` will capture a delimited continuation, because it is not the whole continuation of the program; rather, just the computation initiated by the call to `call-with-continuation-prompt`.

As an example, consider this simple expression:

```
(define c #f)

(* 2
 (+ 3 4
    (call/cc
     (lambda (k)
       (set! c k)
       0)))) ; => 14
```

This code is straightforward, it captures a continuation and stores it in the global binding `c`. The saved continuation looks like this:

```
(* 2 (+ 3 4 _))
```

We can apply the continuation as usual:

```
(c 3) ; => 20
```

To capture part of the continuation we can use a prompt. For instance, if we want capture only `(+ 3 4 _)`:

```
(* 2
  (call-with-continuation-prompt
    (lambda ()
      (+ 3 4
        (call/cc
          (lambda (k)
            (set! c k)
            0))))))) ; => 14

(c 3) ; => 10
```

The `call-with-continuation-prompt` procedure is semantically equivalent to the `TopLevelHandler` previously described as part of the Kawa `call/cc` implementation, and can be expressed with a simple macro:

```
(define-namespace <TLH>
    <gnu.expr.continuations.TopLevelHandler>)

(define (%tlh f)
  (<TLH>:runInTopLevelHandler f))

(define-syntax call-with-continuation-prompt
  (syntax-rules ()
    ((_ f)
     (%tlh (lambda (x) (f))))))
```

Other Scheme implementations, such as Racket or Guile, provides an extended version of this procedure that allows to set prompt tags and handlers. That extended version could be in theory implemented in Kawa modifying `TopLevelHandler` to support custom handlers.

`call-with-continuation-barrier`

Another procedure that we can provide is `call-with-continuation-barrier`. It applies a function with a continuation barrier between the application and the current continuation, then returns the result of the function call. Morover, it do not allow the invocation of continuations that would leave or enter the dynamic extent of the call to `call-with-continuation-barrier`. Such an attempt causes an exception to be thrown.

```
(define-namespace <CH>
   <gnu.expr.continuations.Helpers>)

(define-syntax call-with-continuation-barrier
  (syntax-rules ()
    ((_ f)
     (try-catch (f)
       (cex <CH>:ContinuationException
         (cex:extend (<CH>:ContinuationFrame
                       (lambda (x)
                         (throw (java.lang.Exception
                                  "attempt to cross a
                                   continuation barrier")))))
         (throw cex))))))
```

The macro replaces the call to `call-with-continuation-barrier` with an exception handler that intercepts `ContinuationException`s. The exception handler extends the continuation with a new frame that when invoked throws an exception. Then re-throws the original `ContinuationException` so that the original `call/cc` call is not affected.

If we try the previous example using this time a continuation barrier, we get an error:

```
(* 2
  (call-with-continuation-barrier
    (lambda ()
      (+ 3 4
        (call/cc
          (lambda (k)
            (set! c k)
            0))))))  ; => 14

(c 3)  ; => java.lang.Exception: attempt to cross a
       ;                          continuation barrier
```

shift and reset

I introduced `shift` and `reset` operators and delimited continuations in Chapter 1. `call/cc` can be used to implement those two operators, as shown by Filinsky et al. in [45]. The following code is a port of their SML/NJ implementation:

```
(define (escape f)
  (call/cc (lambda (k)
             (f (lambda x
                  (apply k x))))))

(define mk #f)

(define (abort x) (mk x))

(define (%reset t)
  (escape (lambda (k)
            (let ((m mk))
              (set! mk (lambda (r)
                         (set! mk m)
                         (k r)))
              (abort (t))))))

(define (shift h)
  (escape (lambda (k)
            (abort (h (lambda v
                        (%reset (lambda ()
                                  (apply k v)))))))))
```

```
(define-syntax reset
  (syntax-rules ()
    ((reset exp ...)
     (%reset (lambda () exp ...)))))
```

Native `shift` and `reset`

Although we can implement delimited continuations using `call/cc`, we can avoid unnecessary overhead implementing `shift` and `reset` in Java, modifying the existing `call/cc` implementation.

The `reset` function is semantically similar to the `TopLevelHandler`, while the `shift` can be seen as a kind of `call/cc`. The main difference is that the continuation captured by the `shift` has a limited extent and behaves as an actual function, returning a value. So invoking the continuation inside the `shift` call does not have the effect of escaping from the procedure.

```
public class Shift extends Procedure1 {

    [...]

    public static Object shift(final Procedure receiver)
        throws ContinuationException {
        try {
            // begin unwind the stack
            throw new DelimitedContinuationException();
        } catch (DelimitedContinuationException sce) {
            sce.extend(new ContinuationFrame(receiver));
            throw sce;
        }
    }
}
```

The `Shift` class works like the `CallCC` one, bu it throws a different type of exception, i.e. a `DelimitedContinuationException`, that does not interfere with the `call/cc` calls.

The `Reset` extends `TopLevelHandler` to implement its functionality. In this case there is no need to run the `try/catch` block in a loop, because of the different nature of delimited continuations. As they can be considered regular functions this handler does not need to catch an `ExitException`, it only need to manage the `DelimitedContinuationException` thrown by the `shift` call.

```
public class Reset extends TopLevelHandler {

    [...]
```

```
    public static Object runInTopLevelHandler(Procedure initialFrame)
        throws Throwable {
        try {
            return initialFrame.apply1(null);
        } catch (DelimitedContinuationException dce) {
            final Continuation k = dce.toContinuation();
            return k.resume(k);
        }
    }
}
```

The `DelimitedContinuation` object is different from a `Continuation` in that it does not throw an exception, but it returns a value. Moreover, the apply method reloads the frames inside a `reset` to handle possible future calls of `shift` inside the original outer `reset`.

```
public class DelimitedContinuation extends Continuation {

    [...]

    public Object apply1(final Object val) throws Throwable {
        Procedure1 t = new Procedure1() {

            @Override
            public Object apply1(Object arg1) throws Throwable {
                return reloadFrames(0, frames.size() - 2, val);
            }
        };

        return Reset.runInTopLevelHandler(t);
    }
```

Selective transformation

Using delimited continuations instead of un-delimited ones, gives us the chance to avoid transforming the whole source code. For instance, if we use `reset`/`shift` in a small portion of a program, we can transform only that portion an leave the rest untouched.

The following code is a basic implementation of this idea. The two macros transform the code starting from a `call-with-continuation-prompt` call. The first macro marks the code to be processed by a successive pass. Then the `call/cc-rewrite` macro operates on the syntax tree performing the A-normalization pass and the instrumentation pass on the expression. This gives us continuation-enabled code enclosed in the `call-with-continuation-prompt`.

```
(define-namespace <TLH> <gnu.expr.continuations.TopLevelHandler>)
(define-namespace <ANF> <gnu.expr.ANormalize>)
(define-namespace <FAI> <gnu.expr.FragmentAndInstrument>)
(define-namespace <COMP> <gnu.expr.Compilation>)

(define (%tlh f)
  (<TLH>:runInTopLevelHandler f))

(define-syntax call-with-continuation-prompt
  (syntax-rules ()
    ((_ f)
     (%tlh (call/cc-rewrite (lambda (x) (f)))))))

(define-rewrite-syntax call/cc-rewrite
  (lambda (x)
    (syntax-case x ()
      ((_ sexp)
       (let ((exp (syntax->expression (syntax sexp))))
         (<ANF>:aNormalize exp (<COMP>:getCurrent))
         (<FAI>:fragmentCode exp (<COMP>:getCurrent))
         exp)))))
```

This concept can be further developed, to support nested prompts, and to achieve something similar to what Rompf et al. did in [23] for the Scala compiler.

Higher order functions

To support the capture of continuations inside higher order functions, it is possible to add them, or at least the most common ones, in a module that is transformed for `call/cc` support end included in the compiler. I defined a Scheme version of `map` and `for-each`, which I added in the standard library of Kawa to experiment the applicability of this technique. The module in which those functions are implemented is compiled with the continuations transformation enabled (this can be done using `(module-compile-options full-continuations: #t)`). Moreover, when a Scheme source file is compiled with the full `call/cc` enabled, the compiler replaces the higher order functions with the instrumented version. This allows to capture continuations inside those functions.

Chapter 5

Case studies

"Who controls the past controls the future. He who controls the present controls the past."

George Orwell, 1984

Asynchronous programming: Async and Await

Asynchronous programming is a programming paradigm that facilitates fast and responsive applications. Asynchronous programming is crucial to avoid the inefficiencies caused by blocking activities, such as accesses to the web. Access to a web resource or to a huge database can be slow or delayed. If such an activity is blocked within a synchronous process, the entire application is stuck. You can avoid performance bottlenecks and enhance the responsiveness of your application by using asynchronous programming. In an asynchronous process, the application can continue with other work that does not depend on the resource to be accessed until the potentially blocking task finishes. However, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain. In this section, I propose a syntax similar to the `async/await` construct already introduced in C#, that allows to execute asynchronous tasks during the normal execution of the program.

```
(define (<async-call> <arg>*)
  ... long running operation
      that returns an int ...)

(async (<async-call> <arg>*)
  ... work independent to the
      int result here  ...
  (await <var> ; <- wait for the result
    ... here you can use ...
    ... the result, contained in <var> ...))
```

We will also see how asynchronous programming features can be added to Scheme using coroutines and delimited continuations.

Coroutines

Coroutines are functions that can be paused and later resumed. They are necessary to build lightweight threads because they provide the ability to change execution context. Coroutines are considered challenging to implement on the JVM, as they are usually implemented using bytecode instrumentation. However, first-class continuations makes painless to implement coroutines. They can indeed be obtained with few lines of code in Scheme. The following code is a port of safe-for-space cooperative threads presented by Biagioni et al. in [46], where the code for managing a queue has been omitted for brevity:

```
(define process-queue
           (make-queue))
(define sync-cont #f)

(define (coroutine thunk)
  (enqueue! process-queue thunk))

(define (dispatch)
  (if (null? (car process-queue))
    (when sync-cont
        (sync-cont))
    ((dequeue! process-queue))))


(define (exit)
  (dispatch))

(define (sync)
  (call/cc
   (lambda (k)
     (set! sync-cont k)
     (dispatch))))
```

```
(define (yield)
  (call/cc
   (lambda (parent)
     (coroutine (lambda ()
                  (parent #f)))
     (dispatch))))

(define (thread-activator)
  (call/cc
   (lambda (parent)
     (let ((f (call/cc
                (lambda (fc)
                 (parent fc)))))
       (f)
       (exit)))))

(define (fork f)
  (call/cc
   (lambda (parent)
     (coroutine (lambda ()
                  (parent #f)))
     ((thread-activator) f))))
```

The function `coroutine` establishes a context for running the passed thunk; the `fork` function starts the execution of a new coroutine. The implementation uses an internal prompt (`thread-activator`) to establish the scope of the coroutine. The state of a running coroutine is saved as a function in the queue when doing a `yield`, then the next coroutine in the queue is started by `dispatch`. To end a process, we can call the `exit` function, which calls `dispatch` without saving the current process in the queue. `sync` allows to wait until all the processes are finished.

Control operators like `call/cc` make the implementation of coroutines simpler because one can separate the management of queues from the processes. Coroutines are used in different applications, because they make certain concurrent computations much easier to express and easier to understand, and because, when the number of threads is high, the can give a significant performance improvement over native threads.

Async with coroutines

With the availability of coroutines and `reset`/`shift` we can implement an `async`/`await` expression in Scheme with few lines of code:

```
(define-syntax async
  (syntax-rules (await)
    ((async call during-exp ...
       (await var after-exp ...))
     (let ((var #f))
        (reset
          (shift (lambda (k)
                   (k)
                   (fork (lambda () ; <- start coroutine
                           (set! var call)
                           (exit)))))
          (fork (lambda () during-exp ... (exit))))
        (sync) ; <- wait until all coroutines finish
        after-exp ...))))
```

Consider the following example. We need to execute a time consuming function call, which can be a loop or a recursive function processing some data, but we would like to do something else in the meantime.

```
(define (long-call)
  (let loop ((x 1))
    (if (< x 100)
      (begin (yield)
             (display x)
             (newline)
             (loop (+ x 1)))
      42)))
```

Calling the long call with the `async` syntax it is possible to execute other code in a concurrent way. We can put `(yield)` call inside the loop to suspend the execution and resume the next coroutine in the queue. We do the same in the code to be executed at the same time. The effect is that of running two tasks at the same

time. The `await` keyword allows to wait for the result of the long call, which is bound to the specified variable (`x` in this example).

The logic is implemented using coroutines, the two expressions to be run concurrently are launched using a `fork`, while the result is awaited using `sync`. `reset`/`shift` allows us to delimit the extent of the continuation to be captured, and to change the order of the executed code.

```
(display "start async call")
(newline)
(async (long-call)
       (display "do other things in the meantime...")
       (newline)
       (let loop ((x 0))
         (when (< x 100)
           (begin (yield)
                  (display (- x))
                  (newline)
                  (loop (+ x 1)))))
       (newline)
  (await x
    (display "result -> ")
    (display x)
    (newline)))
```

The above code prints:

```
start async call
do other things in the meantime...
0
1
-1
2
-2
[...]
98
-98
99
-99

result -> 42
```

Async with threads

Using threads instead of coroutine we can avoid adding `(yield)` calls in our code, maintaining the same syntax. Kawa provides a simple interface to create parallel

threads: `(future expression)` creates a new thread that evaluates `expression`, while `(force thread)` waits for the thread's expression to finish executing, and returns the result. Kawa threads are implemented using Java threads.

Thus we can remove `(yield)` calls from our code and redefine the `async`/`await` syntax to use Kawa threads:

```
(define-syntax async
  (syntax-rules (await)
    ((async call during-exp ...
       (await var after-exp ...))
     (let ((var #f))
        (reset
         (shift (lambda (k)
              (set! var (future call)) ; <- start thread
              (k)))
         during-exp ...)
        (set! var (force var)) ; <- wait for result
        after-exp ...))))
```

Now the two tasks are run in parallel, and their printed output is not deterministic:

```
start async call
do other things in the meantime...
0
-1
1
2
-2
[...]
98
99
-98
-99

result -> 42
```

Kawa debugger

Instrumentation allows to suspend the execution of a program, store its state, and resume it, even multiple times. Thus, we can exploit the instrumentation performed to obtain first-class continuations in Kawa to implement debugging features. I extended the technique described in Chapters 3-4 to implement a simple debugger.

When you enable the debugging mode, the compiler instruments each atomic expression with debugging calls, and generates code to store variable bindings in an internal table. When the resulting code runs, it stops at breakpoints and lets you step through the program and inspect variables.

As an example, suppose we need to debug this snippet of code:

```
1 (define (get-first pred lst)
2   (call/cc
3     (lambda (return)
4       (for-each (lambda (x)
5                   (if (pred x)
6                     (return x)))
7                   lst)
8     #f)))
9
10 (get-first negative? '(1 2 3 4 -5 6 7 8 9)) ; => -5
```

We can add a pausing instruction simply calling the `breakpoint` function, as you can see below:

```
...
    (for-each (lambda (x)
                (breakpoint) ; <--
                (if (pred x)
...
```

Once the program is run the execution stops at the breakpoint line, opening a terminal that accepts some predefined commands. The following commands are supported:

command	result
s(tep)	run for one step
c(ontinue)	run until the next breakpoint
p(rint) [var]	print a variable
q(uit)	exit the program

The following listing shows a session of the debugger. In this case the user prints some variable values, then steps forward two times, executing one atomic expression at each step, then continues to stop at the breakpoint at each cycle of the `for-each` until the function returns:

```
### suspended at line 5 ###
> print x
| x: 1
> print return
| return: #<continuation>
> step
### suspended at line 6 ###
### after expression
(Apply line:6:8 (Ref/24/Declaration[applyToArgs/2])
  (Ref/23/Declaration[pred/101])
  (Ref/25/Declaration[x/135]))
###
> step
### suspended at line 5 ###
> print x
| x: 2
> continue
### suspended at line 5 ###
> continue
### suspended at line 5 ###
> print x
| x: 4
> print #all
| get-first: #<procedure get-first>
| pred: #<procedure negative?>
| x: 4
| lst: (1 2 3 4 -5 6 7 8 9)
| return: #<continuation>
### suspended at line 5 ###
> continue
-5
```

Implementation details

The debugger works adding suspension instruction between each atomic expression. After A-normalisation the code is already transformed in a form suitable for instrumentation. During the fragmentation and instrumentation pass, needed by `call/cc`, the syntax tree visitor adds the debug instructions. When the execution reaches a breakpoint call the program is suspended and the user can insert his commands.

The breakpoint call also enables the step mode. Suspension instructions between atomic expressions are disabled during the normal execution, but they are activated when the user gives the `step` command. When the step mode is on, the program stops at each atomic instruction running a simple REPL. When the user gives the `continue` command, the step mode is disabled and the programs can run until the next breakpoint call.

```
(define (breakpoint . args)
  (dbg:enableStepMode)
  (suspend (car args) (cadr args)))

(define (suspend line sourceLine)
  (when dbg:stepMode
    (begin
      (dbg:printInfo line sourceLine (current-output-port))
      (let loop ()
        (let* ((in (read-line))
               (cmds ((in:toString):split " "))
               (cmd (string->symbol (cmds 0))))
          (case cmd
            ((c continue) (dbg:disableStepMode))
            ((p print)
             (if (> cmds:length 1)
               (begin
                 (print (string->symbol (cmds 1)))
                 (loop))))
            ((s step) (void))
            ((q quit exit) (exit))
            (else (display (string-append
                            "unknown command "
                            (cmds 0)))
                  (newline)
                  (loop)))))))))
```

Regarding the debugging instrumentation, the main part is performed in the `visitLetExp` method. I generate a new suspend expression, besides another instruction to add the value of the bind variable to the debugger table at runtime. For each let-bind expression, when the user calls the `print` command, he gets the last value of that variable from the table.

```
protected Expression visitLetExp(LetExp exp, Void ignored) {
    Declaration letDecl = exp.firstDecl();
    Expression continueValue = letDecl.getInitValue();
    String symbol = letDecl.getName();
```

```
if (Compilation.enableDebugger) {
    int lnum = continueValue.getLineNumber();
    String codeLine = continueValue.print();;

    // suspension instruction
    ApplyExp suspend = new ApplyExp(applyRef,
                                    suspendProc,
                                    new QuoteExp(lnum),
                                    new QuoteExp(codeLine));

    // add binding to the debugger binding table
    ApplyExp addVar = new ApplyExp(applyRef,
                                    addBindingProc,
                                    QuoteExp.getInstance(symbol),
                                    new ReferenceExp(letDecl));

    exp.body = new BeginExp(new Expression[]{addVar,
                                             suspend,
                                             exp.body});
}

...
```

Chapter 6

Evaluation

"Extraordinary claims require extraordinary evidence."

Carl Sagan, Encyclopedia Galactica

Transformation overhead

We saw in the previous chapters how we can implement `call/cc` in a JVM targeting compiler, performing a transformation on the whole source to instrument the original code. We would like to know how this global transformation impacts the overall performances of the program when no continuations are captured. We already observed that exception handlers are not expensive on the JVM, but there are other variables to take in consideration. The code fragmentation implies an increase on the number of function calls, which can reduce performance.

I used a set of benchmarks to analyse the behaviour of the running code in the case of both transformed code and non-transformed code. All the benchmarks were executed on an Intel i5 dual-core processor with 4GB of RAM (i5-2410M , 2.30GHz).The operating system was Debian GNU/Linux. The table in Figure 6.1 and the chart in Figure 6.2 show the results.

	Kawa	Kawa (compiled)	Kawa fcc	Kawa fcc (compiled)
fib	11.244	7.496	19.452	16.16
tak	6.636	4.308	10.32	6.956
cpstak	7.984	5.108	10.688	7.048

Figure 6.1: Transformed vs non-transformed code, 10 iterations, values in seconds

The `fib` benchmark runs a simple Fibonacci function with 30 as input. `tak` implements the Takeuchi function and runs it with 18, 12, 6. `cpstak` is a version of

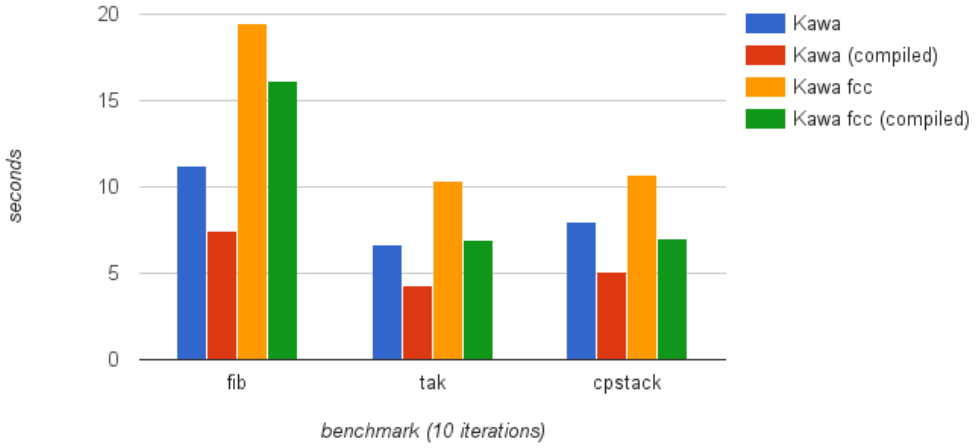


Figure 6.2: Transformed vs non-transformed code, performance comparison

`tak` rewritten in continuation passing style. We can observe that the transformation introduces a considerable overhead, especially in the `fib` benchmark.

To understand from where this overhead comes from, I profiled the execution of the fib benchmark using HPROF, a profiling tool provided by the Java platform [47]. Considering the cpu usage data (Figure 6.3), we can observe that approximately 10% of the cpu time is spent allocating `Proceure` objects (`gnu.mapping.Procedure.<init>` and `gnu.expr.ModuleMethod.<init>`).

rank	method	self time	accum time	count
1	gnu.mapping.Procedure.<init>	4.95%	4.95%	18847828
2	gnu.mapping.ProcedureN.<init>	4.90%	9.85%	18847761
3	gnu.kawa.functions.AddOp.apply2	2.49%	12.34%	2692536
4	gnu.mapping.PropertySet.setSymbol	2.49%	14.83%	9423880
5	gnu.mapping.PropertySet.setProperty	2.46%	17.29%	9423944
6	gnu.kawa.functions.Arithmetic.asIntNum	2.28%	19.57%	8077608
7	gnu.expr.ModuleMethod.<init>	2.25%	21.81%	5385074
8	gnu.expr.ModuleMethod.<init>	2.23%	24.04%	5385074
9	gnu.mapping.PropertySet.<init>	2.03%	26.08%	18847832
10	fib$frame.lambda1continueFragment$X	1.71%	27.79%	2692537

Figure 6.3: Most called Java methods in the `fib` benchmark

We can reach the same conclusions analysing the heap usage. Figure 6.6 shows which objects are more often allocated during the execution of `fib`.

Almost 40% of the heap is used to store object of type `ModuleMethodWithContext`,

rank	name	self	accum	live objects (bytes)	live objs	allocated objs (bytes)	allocated objs
1	gnu.expr.ModuleMethodWithContext	38.45%	38.45%	85949472	2685921	603128352	18847761
2	java.lang.Object[]	33.64%	72.09%	75209400	1343025	527740920	9423945
3	fib$frame	5.49%	77.58%	12278368	383699	86161184	2692537
4	fib$frame0	5.49%	83.07%	12278368	383699	86161184	2692537
5	fib$frame2	3.43%	86.50%	7674120	191853	53850720	1346268
6	fib$frame1	2.75%	89.25%	6139328	191854	43080576	1346268
7	fib$frame3	2.75%	92.00%	6139296	191853	43080576	1346268
8	java.lang.Object[]	2.06%	94.06%	4604496	191854	32310432	1346268
9	java.lang.Object[]	2.06%	96.12%	4604280	191845	32310432	1346268
10	java.lang.Object[]	2.06%	98.18%	4604016	191834	32310432	1346268

Figure 6.4: Most allocated Java object during the execution of the `fib` benchmark

	Kawa	Kawa (compiled)	Kawa fcc	Kawa fcc (compiled)
fib	55232	41484	362060	360428
tak	47908	41364	58308	52152
cpstak	49404	40868	58516	55644

Figure 6.5: memory usage in transformed vs non-transformed code, values in Kbytes

that is the runtime object in which closures are allocated. This is not unexpected, as the transformed code is fragmented in a set of closures. However, this suggest that a possible improvement for the technique can be obtained optimising closure allocation.

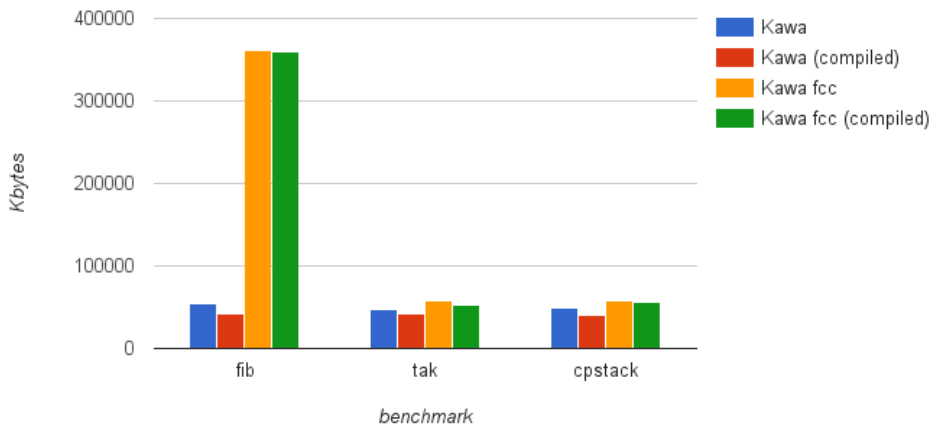


Figure 6.6: Transformed vs non-transformed code, memory usage comparison

`call/cc` performance

I tested the new `call/cc` implementation on five continuation-intensive benchmarks. `fibc` is a variation of `fib` with continuations. The `loop2` benchmark corresponds to a non-local-exit scenario in which a tight loop repeatedly throws to the same continuation. The `ctak` benchmark is a continuation-intensive variation of the call-intensive `tak` benchmark. The `ctak` benchmark captures a continuation on every procedure call and throws a continuation on every return. In addition to `fibc` `loop2` and `ctak`, already used in [19], I used a benchmark based on coroutines, and another implementing a generator.

I compared the modified version of Kawa with other Scheme implementations with an interpreter or JIT compiler, targeting either native machine code or an internal VM:

- Petite Chez Scheme is a sibling version of Chez Scheme, a proprietary Scheme implementation. Petite is a threaded interpreter and can be used free of charge.
- Chicken is a Scheme to C compiler, but also an interpreter.
- Gambit is a Scheme implementation, which has both and interpreter and a compiler that produces C code.
- Guile is an interpreter and compiler for Scheme, using a virtual machine that executes a portable instruction set generated by its optimizing compiler, and integrates very easily with C and C++ application code.
- Racket is a programming language based on standard Scheme, but includes way more features in the base language. It also offers an IDE and a large number of built in libraries and tools.
- SISC is a Scheme interpreter written in Java, and running on the JVM. SISC is also the only other JVM Scheme supporting `call/cc`.

	Petite Chez 8.4	Chicken 4.9.0.1	Gambit 4.7.6	Racket 6.1.1	Guile 2.0.11	SISC 1.16.6	Kawa 2.0.1 fcc
fibc	2.064	8.868	5.06	10.56	157.704	35.724	31.164
ctak	0.912	1.844	0.608	5.4	22.108	23.888	17.276
loop2	1.828	2.956	4.024	9.508	95.1	32.816	17.04
coroutines	0.836	1.068	0.664	3.48	15.136	22.128	24.628
generators	0.792	0.036	3.116	4.12	0.288	15.004	9.936

Figure 6.7: Capturing benchmark (interpreted code), 10 iterations, values in seconds

Some of the Scheme implementations introduced above can pre-compile code to a bytecode or binary format, which can be later executed without paying the cost for translation. Figures 6.13 and 6.14 compares the execution time of code compiled by five compilers, including the modified version of Kawa.

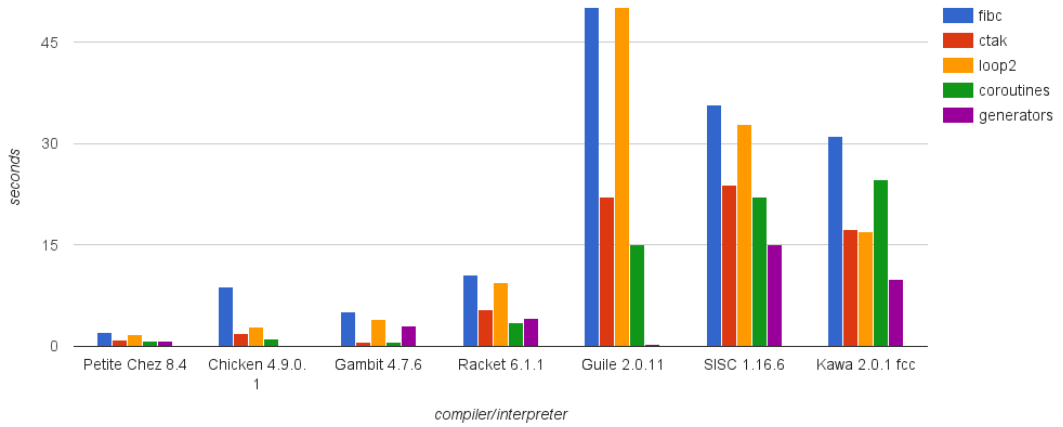


Figure 6.8: Capturing benchmark (interpreted code), 10 iterations

	Chicken 4.9.0.1	Gambit 4.7.6	Racket 6.1.1	Guile 2.0.11	Kawa 2.0.1 fcc
fibc	0.528	0.564	10.176	48.184	19.388
ctak	0.128	0.12	4.232	10.416	11.424
loop2	0.572	0.796	9.288	7.54	10.888
coroutines	0.14	0.064	3.428	5.264	15.62
generators	0.02	0.02	2.24	0.324	4.328

Figure 6.9: Capturing benchmark (pre-compiled code), 10 iterations, values in seconds

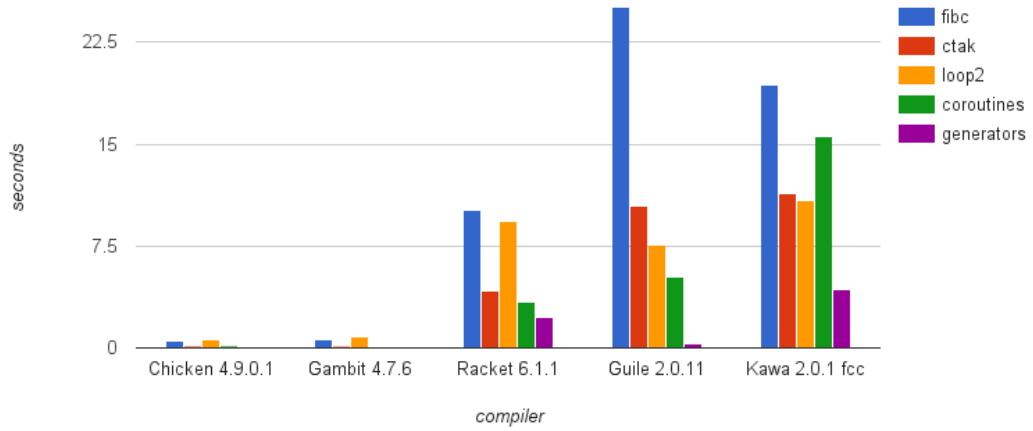


Figure 6.10: Capturing benchmark (pre-compiled code), 10 iterations

Looking at the benchmarks' outcome we can see that Kawa with first-class continuations (Kawa fcc), despite the overhead we measured in the previous section, performs slightly better then SISC. As expected, Kawa fcc performances are far from the Scheme to C compilers, however, when compared with Guile and Racket they are within the same order of magnitude.

`call/cc` memory usage

I measured peak memory usage of the same five benchmarks introduced in the performance section, testing the same range of compilers. This time Kawa fcc performs similarly to SISC, except for the `fibc` benchmark. Kawa fcc also uses a similar amount of memory similar to Racket in the `coroutines`, `generators` and `ctak` benchmarks. Chez and Scheme to C compilers have performances unreachable for implementations using a VM, both in interpreted and compiled modes.

	Petite Chez 8.4	Chicken 4.9.0.1	Gambit 4.7.6	Racket 6.1.1	Guile 2.0.11	SISC 1.16.6	Kawa 2.0.1 fcc
fibc	26256	11572	9028	68508	8276	202228	304208
ctak	26268	10312	8496	58492	7824	112328	79500
loop2	26312	6776	6304	55876	7828	152016	170016
coroutines	26260	9508	8500	119948	230128	90624	85956
generators	26260	6468	32260	54436	8320	86196	60220

Figure 6.11: Peak memory usage (interpreted code), 10 iterations, values in Kbytes

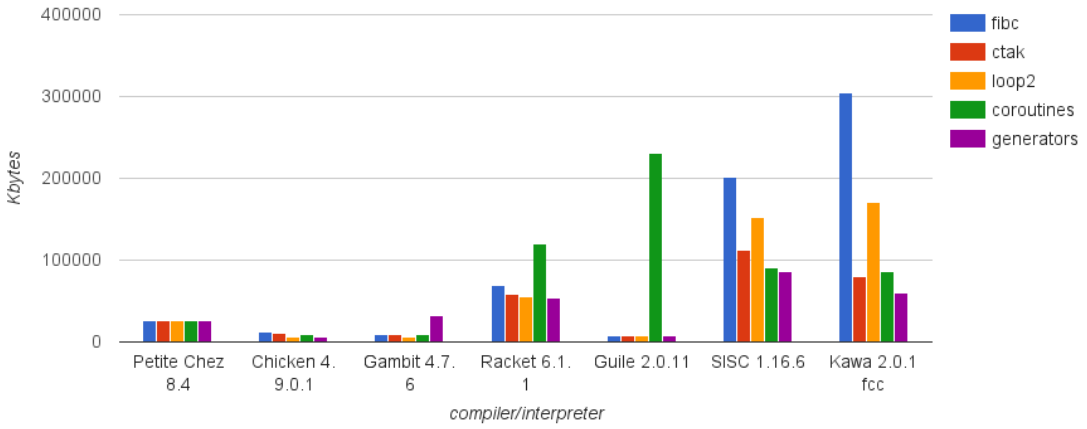


Figure 6.12: Peak memory usage (interpreted code), 10 iterations

I repeated the same benchmarks using pre-compiled code. However, with relation to memory usage, the differences between interpreted vs compiled code is negligible.

	Chicken 4.9.0.1	Gambit 4.7.6	Racket 6.1.1	Guile 2.0.11	Kawa 2.0.1 fcc
fibc	6128	8256	68300	11036	294212
ctak	5916	8252	58164	11704	70648
loop2	5840	6152	55632	7924	162968
coroutines	6352	8224	120880	217440	72104
generators	5776	6124	44876	7952	41452

Figure 6.13: Peak memory usage (pre-compiled code), 10 iterations, values in Kbytes

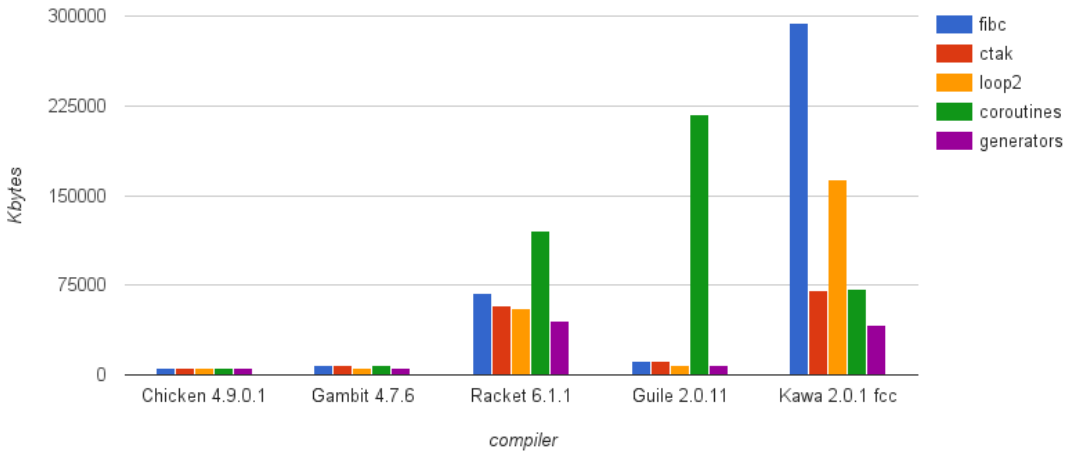


Figure 6.14: Peak memory usage (pre-compiled code), 10 iterations

Code size

We saw in Chapter 3 that we expect an increase in code size proportional to the number of code fragments, so we want to measure the actual difference in size between a regular class file and an instrumented one. Figure 6.15 shows a comparison of regular code and transformed code.

	Kawa 2.0.1	Kawa 2.0.1 fcc
fib	2360	19165
tak	2632	26921
cpstak	10886	31131
srfi34	28089	86863

Figure 6.15: Code size comparison, values in bytes

We can observe that the size of transformed code can be 10 times larger than the code compiled without first-class continuations enabled. Even if the code size increase is proportional to the number of fragments, the difference in size is significant. This indicates that would be better to limit the use of transformed code to modules that needs `call/cc`, and use `call/cc` enabled code in combination with non-transformed code.

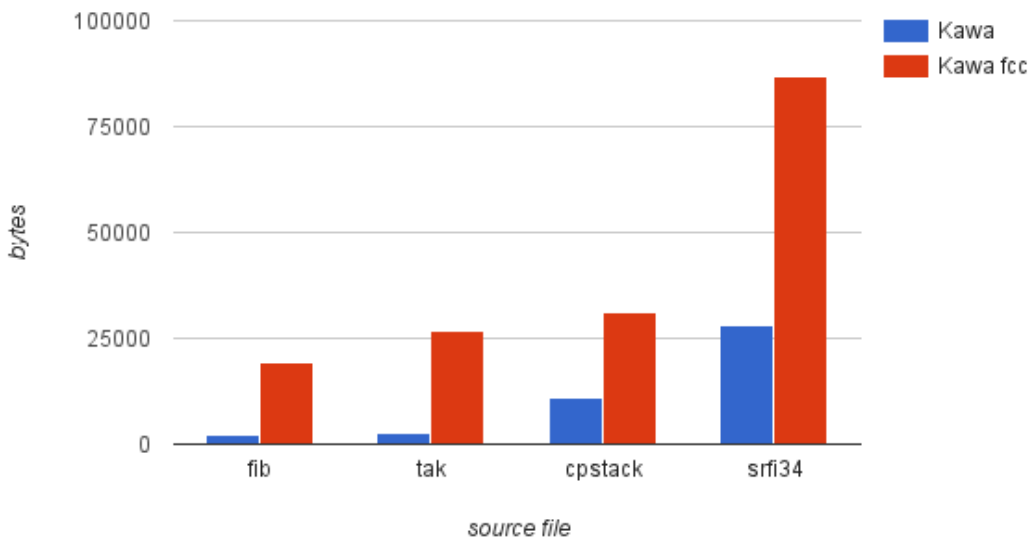


Figure 6.16: Size of compiled classes in bytes

Chapter 7

Conclusions and future work

This dissertation has presented an implementation of the `call/cc` control operator in Kawa, a Scheme compiler targeting the JVM. Although the problem was delineated in some works in literature, there was not a compiler providing first-class continuations on the JVM in terms of `call/cc`.

I developed a variant of generalised stack inspection in the Kawa compiler, addressing the problem of defining an A-normalisation algorithm for the Kawa super-set of Scheme, and realising a fragmentation and instrumentation pass using the existing Kawa framework. The whole transformation has been designed to be optional and separated from the existing passes, so that it does not add unnecessary overhead to modules without continuations.

I explored variations of the technique to implement other control operators, such as `shift`/`reset` and prompts, as well as continuation barriers. Moreover, the two passes are flexible enough that could be used on a portion of the syntax tree, instead of the entire program.

I showed the opportunities opened by the availability of `call/cc`, developing a syntax for asynchronous programming, and exploiting the A-normalisation of the syntax tree to create a simple debugger.

The evaluation of performance and memory usage revealed that this technique can be a valid alternative to heap-based implementations of `call/cc`. Benchmarks also showed that the bottleneck of the technique is not exception handling, but closure allocation, leaving room for improvement.

Future work

This work can be further developed in several interesting directions. I will outline a few possible applications and extensions, which can be based on this contribution to implement new features and obtain improvements in other programming languages or framework.

- Variants of this technique can be employed in other JVM languages to implement first-class delimited and un-delimited continuations and control operators. Languages like Clojure, JRuby, Groovy, Bigloo and others could take advantage of this work.

- The transformation described in the previous pages uses an intermediate A-normalisation pass. ANF has been related in other works to Static Single Assignment (SSA) form [48]. It is possible to exploit the formal properties of ANF to implement in the compiler many optimisation present in literature.

- In the past few years several frameworks for concurrency appeared on the Java scene. Many of them use bytecode instrumentation to implement coroutines and provide lightweight threads with low memory and task-switching overhead. Quasar [49], for instance, deliver the actor model on the JVM using bytecode instrumentation. The transformation presented in this document can give an alternative for those frameworks that aim to provide a concurrency API for Java or JVM languages.

- Research has been done on the use of continuations in the context of web applications [50, 51]. The support for first-class continuations developed in the context of this thesis, can be utilised to implement continuation based web frameworks in Kawa or in Java.

- The research field of Dynamic Software Updating (DSU) pertains to upgrading programs while they are running [52, 53]. Different approaches for DSU has been developed, nevertheless it is not currently widely used in industry. Some of the approaches use a stack reconstruction technique similar in many aspects to the `call/cc` implementation described in this dissertation [54]. Future work can start from the achievements of this work to explore an alternative implementation of DSU on the JVM.

References

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm. [Accessed: 14-Jun-2015]

[2] "TIOBE Software: Tiobe Index." [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. [Accessed: 11-Jun-2015]

[3] "Java (programming language) - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Java_(programming_language). [Accessed: 13-Jun-2015]

[4] "Lambda Expressions (The Java™ Tutorials)." Oracle [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html. [Accessed: 15-Jun-2015]

[5] M. Fusco, "Why We Need Lambda Expressions in Java | Javalobby." 27-Mar-2013 [Online]. Available: http://java.dzone.com/articles/why-we-need-lambda-expressions. [Accessed: 13-Jun-2015]

[6] "Java virtual machine - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine. [Accessed: 13-Jun-2015]

[7] "Alternative Languages for the JVM." [Online]. Available: http://www.oracle.com/technetwork/articles/java/architect-languages-2266279.html. [Accessed: 14-Jun-2015]

[8] "Scheme (programming language) - Wikipedia, the free encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Scheme_(programming_language). [Accessed: 15-Jun-2015]

[9] R. K. Dybvig, *The Scheme programming language, Fourth Edition.* 2009 [Online]. Available: http://www.scheme.com/tspl4/

[10] R. K. Dybvig, "Three implementation models for scheme," PhD thesis, University of North Carolina at Chapel Hill, 1987 [Online]. Available: http://www.cs.indiana.edu/~dyb/pubs/3imp.pdf

[11] M. Might, "Continuations by example." [Online]. Available: http://goo.gl/ECRA5r. [Accessed: 15-Jun-2015]

[12] D. Martins, "Why Are Continuations So Darn Cool?" [Online]. Available: http://danielmartins.ninja/posts/why-are-continuations-so-darn-cool.html. [Accessed: 15-Jun-2015]

[13] D. Madore, “A page about call/cc.” [Online]. Available: http://www.madore.org/~david/computers/callcc.html. [Accessed: 15-Jun-2015]

[14] K. Asai and O. Kiselyov, *Introduction to Programming with Shift and Reset*. 2011 [Online]. Available: http://www.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf

[15] O. Kiselyov and C.-c. Shan, “Delimited continuations in operating systems,” in *Modeling and Using Context*, Springer, 2007, pp. 291–302 [Online]. Available: http://www.okmij.org/ftp/continuations/ZFS/context-OS.pdf

[16] “Continuations - Racket Documentation.” [Online]. Available: http://docs.racket-lang.org/reference/cont.html#(part._.Classical_.Control_.Operators). [Accessed: 14-Jun-2015]

[17] “Delimited continuation - Wikipedia, the free encyclopedia.” [Online]. Available: https://en.wikipedia.org/wiki/Delimited_continuation. [Accessed: 14-Jun-2015]

[18] “Kawa: The Kawa Scheme language.” [Online]. Available: http://www.gnu.org/software/kawa/index.html. [Accessed: 15-Jun-2015]

[19] W. D. Clinger, A. H. Hartheimer, and E. M. Ost, “Implementation strategies for first-class continuations,” *Higher-Order and Symbolic Computation*, vol. 12, no. 1, pp. 7–45, 1999 [Online]. Available: http://link.springer.com/article/10.1023/A:1010016816429

[20] S. G. Miller, “SISC: A complete scheme interpreter in java,” Technical Report, Jan, 2002 [Online]. Available: http://sisc-scheme.org/sisc.pdf

[21] A. W. Appel, *Compiling with continuations*. Cambridge University Press, 2006.

[22] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin, *Orbit: An optimizing compiler for Scheme*, vol. 21. ACM, 1986 [Online]. Available: http://dl.acm.org/citation.cfm?id=13333

[23] T. Rompf, I. Maier, and M. Odersky, “Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform,” in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming - ICFP ’09*, 2009, p. 317 [Online]. Available: http://dl.acm.org/citation.cfm?doid=1596550.1596596

[24] J. McBeath, “Delimited Continuations.” August-2010 [Online]. Available: http://jim-mcbeath.blogspot.co.uk/2010/08/delimited-continuations.html

[25] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen, “Continuations from generalized stack inspection,” in *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming - ICFP ’05*, 2005, p. 216 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1086365.1086393

[26] T. Sekiguchi, T. Sakamoto, and A. Yonezawa, “Advances in Exception Handling Techniques,” A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, Eds. Springer Berlin Heidelberg, 2001, pp. 217–233 [Online]. Available: http://dx.doi.org/10.1007/3-540-45407-1_14

[27] W. Tao, "A portable mechanism for thread persistence and migration," PhD thesis, School of Computing, University of Utah, 2001 [Online]. Available: http://www.cs.utah.edu/~tao/research/

[28] F. Loitsch, *Exceptional continuations in JavaScript*. 2007 [Online]. Available: http://repository.readscheme.org/ftp/papers/sw2007/04-loitsch.pdf

[29] J. Marshall, "An Unexceptional Implementation of First-Class Continuations," in *Proceedings of the 2009 International Lisp Conference*, 2009, pp. 36–40 [Online]. Available: https://drive.google.com/file/d/0B4zj6rF-PmISMTNjYWVhYmYtZTQ1MC00YTg4LWFiYTQtYzJiZjQwMzQyMjA1/view?ddrp=1&hl=en

[30] S. Srinivasan, *A thread of one's own*, vol. 4. 2006 [Online]. Available: http://malhar.net/sriram/kilim/thread_of_ones_own.pdf

[31] L. Bolton, "Kilim: a server framework with lightweight actors, isolation types, and zero-copy messaging," *The Contemporary Pacific*, vol. 12, no. 2, pp. 561–563, 2000 [Online]. Available: http://muse.jhu.edu/content/crossref/journals/contemporary/_pacific/v012/12.2bolton.html

[32] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen, "A Technique for Implementing First-Class Continuations." 2005 [Online]. Available: http://www.ccs.neu.edu/racket/pubs/stackhack4.html

[33] J. Rose, "Longjumps Considered Inexpensive." 10-May-2007 [Online]. Available: https://blogs.oracle.com/jrose/entry/longjumps_considered_inexpensive. [Accessed: 09-Jun-2015]

[34] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation - PLDI '93*, 1993, pp. 237–247 [Online]. Available: http://portal.acm.org/citation.cfm?doid=155090.155113

[35] "Commons Javaflow - Overview." [Online]. Available: http://commons.apache.org/sandbox/commons-javaflow/. [Accessed: 09-Jun-2015]

[36] L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose, "Lazy continuations for Java virtual machines," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java - PPPJ '09*, 2009, p. 143 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1596655.1596679

[37] "RIFE : About." [Online]. Available: http://rifers.org/. [Accessed: 09-Jun-2015]

[38] A. Begel, J. MacDonald, and M. Shilman, "PicoThreads: Lightweight threads in Java," *Department of Electrical Engineering & Computer Sciences University of California, Berkeley*, 2000 [Online]. Available: http://research.microsoft.com/en-us/um/people/abegel/cs262/picothreads.pdf

[39] M. Mann, "Continuations Library." [Online]. Available: http://www.matthiasmann.de/content/view/24/26/. [Accessed: 09-Jun-2015]

[40] P. Bothner, "Kawa internals: Compiling Scheme to Java." 17-November-1998 [Online]. Available: http://www.gnu.org/software/kawa/internals/index.html

[41] M. Might, "A-Normalization: Why and How." [Online]. Available: http://matt.might.net/articles/a-normalization/. [Accessed: 17-Jun-2015]

[42] "OpenJDK: jmh." [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/. [Accessed: 23-Jun-2015]

[43] J. Ponge, "Avoiding Benchmarking Pitfalls on the JVM." [Online]. Available: http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html. [Accessed: 23-Jun-2015]

[44] "1.1 Evaluation Model." [Online]. Available: http://docs.racket-lang.org/reference/eval-model.html#%28part._prompt-model%29. [Accessed: 28-Jun-2015]

[45] A. Filinski, "Representing monads," in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*, 1994, pp. 446–457 [Online]. Available: http://portal.acm.org/citation.cfm?doid=174675.178047

[46] E. Biagioni, K. Cline, P. Lee, C. Okasaki, and C. Stone, "Safe-for-space threads in Standard ML," *Higher-Order and Symbolic Computation*, vol. 11, no. 2, pp. 209–225, 1998 [Online]. Available: http://link.springer.com/article/10.1023/A:1010016600604

[47] "HPROF: A Heap/CPU Profiling Tool." Oracle [Online]. Available: http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html. [Accessed: 26-Jun-2015]

[48] M. M. Chakravarty, G. Keller, and P. Zadarnowski, "A Functional Perspective on SSA Optimisation Algorithms," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 2, pp. 347–361, Apr 2004 [Online]. Available: http://dx.doi.org/10.1016/S1571-0661(05)82596-4

[49] "Parallel Universe." Parallel Universe [Online]. Available: http://blog.paralleluniverse.co/2015/05/21/quasar-vs-akka/. [Accessed: 29-Jun-2015]

[50] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen, "Automatically Restructuring Programs for the Web," *Automated Software Engineering*, vol. 11, no. 4, pp. 337–364, Oct 2004 [Online]. Available: http://dx.doi.org/10.1023/B:AUSE.0000038936.09009.69

[51] C. Queinnec, "Continuations and Web Servers," vol. 17, no. 4, pp. 277–295, Dec 2004 [Online]. Available: http://dx.doi.org/10.1007/s10990-004-4866-z

[52] A. R. Gregersen, M. Rasmussen, and B. N. Jørgensen, "State of the Art of Dynamic Software Updating in Java," in *Software Technologies*, Springer, 2014, pp. 99–113 [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-3-662-44920-2_7

[53] K. Makris and R. A. Bazzi, *Immediate Multi-Threaded Dynamic Software*

Updates Using Stack Reconstruction., vol. 2009. 2009 [Online]. Available: http://static.usenix.org/legacy/events/usenix09/tech/full_papers/makris/makris.pdf

[54] J. Buisson and F. Dagnat, "Introspecting continuations in order to update active code," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades - HotSWUp '08*, 2008, p. 1 [Online]. Available: http://portal.acm.org/citation.cfm?doid=1490283.1490289