An Interpreter for `JLambda`

By
David Porter

A thesis submitted for the degree of
Bachelor of Computing Science (Honours)
of The University of New England
December 2004

DECLARATION

I certify that the substance of this thesis has not already been submitted for any degree and is not currently being submitted for any other degree.

I certify that to the best of my knowledge, any help received in preparing this thesis, and all sources used, have been acknowledged in this thesis.

. . . .

Acknowledgements

I would like to thank my Honours supervisor, Ian Mason, for his enthusiasm, help, and timely encouragement during this work. His patience, despite my many questions, is greatly appreciated.

I must also thank Carolyn Talcott for her invaluable insights and guidance in this work, and for her generosity and dedication in suggesting improvements to drafts of this thesis.

Finally, I am forever indebted to Sarah for her understanding, endless forbearance, and encouragement when it was most required.

Abstract

In this thesis an interpreter for `JLambda`, an untyped Scheme-like lexically scoped language, is designed and implemented in the Java programming language. We describe the difficulties that arise when a non-tail recursive implementation language is chosen to implement an interpreter for a tail recursive target language. We show that these difficulties can be overcome by applying the continuation passing style transformation and the register machine transformation to the interpreter implementation.

We also explain several modifications to the implementation of the interpreter that yield an increase in the interpreter's execution speed. We demonstrate how those improvements can be realised by exploiting various features of the Java language.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents techniques for using the Java [8] programming language to implement an efficient interpreter for the `JLambda` language. The `JLambda` language is a component of the IOP[10] project, which is aimed at developing an infrastructure for allowing formal reasoning tools to interoperate, and to interact with users in a transparent and elucidating graphical fashion.

The `JLambda` language is an untyped Scheme-like [9] lexically scoped interpreted language that provides a runtime interface to the Java class library. `JLambda` makes very heavy use of Java's built-in reflective capabilities. It is designed to be efficient and expressive enough to enable full and faithful use of any built-in Java classes.

In this thesis we describe the difficulties that arise when one chooses a non-tail recursive implementation language for an interpreter of a tail recursive target language, and explain how those difficulties may be overcome. We also explain several modifications to the interpreter's implementation that yield an increase in the interpreter's execution speed. We demonstrate how those improvements can be realised by exploiting the features of the Java programming language.

Chapter 2 outlines other work that is related to the work presented here. We list three Scheme interpreters that are implemented in Java, and describe the ways in which the intentions of those projects differ from ours.

A detailed description of `JLambda` can be found in Chapter 3. We provide an overview of the language and a description of its syntax and semantics. We also explain the the features provided for run-time access to the Java language facilities.

A example of a complete `JLambda` program is presented it Chapter 4. The program, `clicker`, highlights many of the interesting features of `JLambda`, giving particular attention to the creation and manipulation of Java objects.

In Chapter 5, we begin by presenting a straight-forward recursive implementation, in Java, of an interpreter for the `JLambda` language. We then demonstrate that although a recursive implementation has the virtues of being simple to implement and easy to understand, the interpreter we have constructed cannot properly evaluate all `JLambda` expressions. Finally, we explain why these difficulties are consequences of choosing as the implementation language a language that is not tail recursive.

A solution to these difficulties is provided in Chapter 6. We use the continuation-passing transformation followed by the register-machine transformation to convert the interpreter from one that has recursive execution to one that has iterative execution. The chapter contains a detailed explanation of how we implement these techniques in Java.

Chapter 7 presents four modifications to the implementation of the interpreter, all with the goal of increasing the interpreter's execution speed. We demonstrate the implementation of each of the modifications, and create a new version of the interpreter for each of the modifications.

Finally, in Chapter 8 we evaluate the modifications to the interpreter's implementation that were made in Chapter 7. Two benchmark programs are introduced, and their use in comparing the execution speeds of the four interpreters is explained. We then present and examine the results of running each of the interpreters with the benchmark programs.

Chapter 2

Related Work

There are several existing Scheme interpreters that use Java as their implementation language. Three well-known examples are Sisc [11], Kawa [4], and JScheme [2]. Although all three of these projects aim to produce a Java-based implementation of the Scheme language, they have each have unique characteristics.

The objective of the Sisc is to provide a lightweight, platform-independent Scheme system whose primary goal is rapid execution of the complete R5RS and future Scheme standards. SISC uses modern interpretation techniques, and outperforms all existing Java Virtual Machine interpreters, often by more than an order of magnitude.

Kawa is a Scheme environment, written in Java, and that compiles Scheme code into the byte-code instructions of the Java Virtual Machine. Kawa is a full Scheme implementation that supports all of the required and optional features of the R5RS Scheme standard except proper tail recursion and unrestricted continuations.

The JScheme project is a dialect of Scheme which features a simple and comprehensive interface to Java. JScheme allows one to access all methods, constructors, and fields for any Java classes by name; it provides access to Java literals, Java scalar operations, Java threads, and Java exception handling; and all of the features of the R4RS Scheme standard, with two exceptions: continuations are only partially supported and strings are immutable.

The primary motivations of all three of these projects differ from those of `JLambda`: whereas Sisc, Kawa and JScheme all seek to provide a complete and faithful implementation of the Scheme language, `JLambda` does not. Instead, `JLambda`'s purpose is to provide a run-time interface to the Java classes, and to some specific classes of the IOP project, of which `JLambda` is a component.

There are several important features of Scheme that `JLambda` does not implement; for example, `JLambda` has no support for first-class continuations or macros. A further difference between `JLambda` and the Scheme implementation listed above, is that `JLambda` does not attempt to implement the Scheme data types, such as symbols and pairs, and the various number types, but instead bases its types on the underlying Java types.

2.1 A Comparison of `JLambda` and JScheme

Of the three projects mentioned, JScheme appears to be closest in spirit to `JLambda`. In this section we compare the interface to Java provided by JScheme to that provided by `JLambda`.

The interface to Java provided by JScheme is called the *Javadot notation*. This notation provides access to all Java classes, constructors, methods, and fields on the classpath. Table 2.1 provides some examples of this notation.

Syntax	Type of Member	Example
"." at the end	constructor	`(Font. NAME STYLE SIZE)`
"." at the beginning	instance member	`(.setFont COMP FONT)`
"." only in the middle	static member	`(Math.round 123.458)`
".class" suffix	Java class	`Font.class`
"$" at the end	static field	`Font.BOLD$`
"$" at the beginning	package-less class	`$Foo.class`

Table 2.1: Examples of JScheme's Javadot notation

Clearly, JScheme's Java interface is designed to closely follow Java syntax. In contrast, the Java interface provided by `JLambda` is intended to follow Scheme syntax.

`JLambda` and JScheme also differ in the support they provide for constructing interactive graphical objects. JScheme defines some helper classes for easily specifying event handlers. The JScheme expression

```
(Listener. (lambda (e) EXPR))
```

returns an object that implements all of the Java Listener interfaces. The action for all methods of these interfaces is to call the lambda expression on the event.

In `JLambda`, however, event listeners may be registered only for instances of classes in the `glyphics` class hierarchy, which is also a component of the IOP project, and which is described in Chapter 4.

More generally, JScheme provides sufficient language facilities for constructing any interactive graphical Java object. In `JLambda`, however, interactive graphical objects must be constructed using the `glyphics` class hierarchy.

Chapter 3

The JLambda Language.

The Scheme-like `JLambda` language is a call-by-value left to right evaluation ordering, lexically scoped language with closures. It has exactly the same underlying primitive data types as Java, and access to all of Java's built in packages and classes.

A `JLambda` expression is either a string of characters or a List.

3.0.1 Variables

A string of characters is interpreted as a variable, and is evaluated as follows: First see if it is a static Java field e.g.

```
java.awt.Color.black,
```

or

```
java.awt.geom.GeneralPath.WIND_EVEN_ODD,
```

if it is, return the current value of that field. Next see if it is bound in the current lexical environment, if it is return its current value. Otherwise look to see if it is bound in the current global environment, if it is return its current value, else throw an unbound variable exception.

3.0.2 Definitions

Global definitions are made using the `define` form. We provide the usual two Scheme versions. The more general form simply binds the value of `<exp>` to the identifier `<name>` (which is not evaluated):

```
(define <name>  <exp>)
```

The more specific function or closure definition version is

```
(define <name> (<param1> ... <paramN>)  <form>)
```

which binds the name `<name>` to the corresponding closure, in the global environment. Note that the latter is just an abbreviation for a variant of the former form. Namely

```
(define <name> (lambda (<param1> ... <paramN>)  <form>))
```

We will discuss closures, and lambda expressions shortly, along with the lexical binding form `let`.

3.0.3 Primitive Data

Primitive data is represented by a primitive data expression, which is a list of length two:

```
(<tag>  <exp>)
```

Neither the `<tag>` position nor the `<exp>` is evaluated. The `<tag>` should be the name of one of Java's primitive data types:

```
 boolean  byte  double  char  float  int  long  short
```

while `<exp>` should be a sequence of characters. It is parsed as the appropriate data, for example `(int 10)` will be parsed as the number `10`, a Java `int`, while `(char 'C')` and `(boolean false)` will be parsed as Java's character `C`, and boolean `false`, respectively.

If parsing as such fails, then it is the assigned the default zero value of that Java data type (`false` in the case of a `boolean`), and a warning to standard error is issued.

The usual literals can be used within these primitive data expressions, a representative sample is given here.

```
(char 88)         ; evaluates to X
(char 'X')        ; evaluates to X
(char '\130')     ; evaluates to X
(char '\u0058')   ; evaluates to X
(char '\n')       ; evaluates to a new line
(byte 127)        ; evaluates to 127
```

```
(byte 128)         ; evaluates to 0 and an error is reported
(int 123456)       ; evaluates to 123456
(int 123456.7)     ; evaluates to 0 and an error is reported
(float 2.5e+27)    ; evaluates to 2.5E27
(boolean true)     ; evaluates to true
(boolean 0)        ; evaluates to false
```

It should be pointed out that unadorned strings that consist solely of digits, for example `1234567890`, are bonafide variables, and can be bound both lexically and globally. To regard them as numbers they need to be wrapped in a primitive data expression. For example, there is nothing wrong with initializing the global environment via the following sequence of expressions.

```
(define 0 (int 0))
(define 1 (int 1))
...
(define 100 (int 100))
```

3.0.4 Numeric Operations

The usual arithmetic operations are provided. These work on numbers and `char`s, *not* `boolean`s. These follow the usual Java contortions, see section 5.6, pages 74–76 of [8]. The expressions are evaluated, they are both widened to `int`s if they are smaller. If one is bigger than an `int` they are both widened to that type, then the operation is performed.

```
(- <exp>)
(- <exp> <exp>)
(* <exp> <exp>)
(+ <exp> <exp>)
(/ <exp> <exp>)
(% <exp> <exp>)
```

To complete the arithmetic operations we provide a corresponding narrowing operation:

```
(narrow <exp> <exp>)
```

This performs one of Java's 23 primitive narrowing conversions (See section 5.1.3, page 55, of [8]). The expressions are evaluated in left to right order. The second `<exp>` should evaluate to one of the strings: `byte`, `short`, `char`, `int`, `long`, or `float`. Indicating the desired primitive data type. The first expression should evaluate to a primitive numeric type (including `char`). The usual loss of precision occurs.

3.0.5 Boolean Relations

We provide the usual binary relations on numbers:

```
(<  <exp> <exp>)
(>  <exp> <exp>)
(<= <exp> <exp>)
(>= <exp> <exp>)
```

Non numeric arguments will cause an exception to be thrown.

Equality is a boolean relation defined on all types of data:

```
(= <exp> <exp>)
```

The expressions are evaluated in left to right order. The equality expression then returns `true` if either both values are `null`, or neither are `null` and both are booleans of the same value, or both are characters of the same value, or both are numbers and are equal, or the second object satisfies the first Object's `equals` method.

There is an inequality expression:

```
(!= <exp> <exp>)
```

which is the same as `(not (= <exp> <exp>))`.

3.0.6 Arrays

Arrays may be constructed and manipulated in a direct manner.

Array Construction

There are two forms of array construction, one corresponds to making an an empty array of a particular size, the other corresponds to making an array, and assigning its contents. A *zeroed* array of a particular length is constructed via:

```
(mkarray  <type>  <exp>)
```

whereas an array is constructed *and assigned* via:

```
(array  <type>   <exp_1> .... <exp_N>)
```

In both cases `<type>` is either a primitive data tag or else the full name of a Java class, `<type>` is not evaluated). In the `mkarray` case the expression `<exp>` should evaluate to an integer expression (or something smaller), and an array of that length is constructed and zeroed, according to the nature of `<type>` in the usual Java way. In the `array` case of an array of of length `N` is constructed. There can be zero or more elements `<exp_i>`, and they should all be of the appropriate type. The contents of the constructed array are initialized to be the values of the `<exp_i>`, evaluated from left to right. Each of the values of the expressions should be able to be considered as an element of the class or type represented by the `<tag>`. `null` is allowable in the case that the `<tag>` names a Java class. Widening is allowable in the case of primitive data, as is upcasting to a interface or superclass in the case that the `<tag>` names a Java class. Otherwise an error will be reported.

In the following examples `a0` will be an array of length `88`, that contains that many `false`s, `a1` will name an array of integers of length `3`, whereas `a2` will name an array of objects of length `1`.

```
(define a0 (mkarray int (char 'X')))
(define a1 (array int (char 'X') (byte 3) (short 7)))
(define a2 (array java.lang.Object java.lang.System.err))
(define a3 (array [I a1 a1)))
(define a4 (array [Ljava.lang.Object; a2 a2 a2))
```

To make an array of arrays one must use Java's quaint array type naming conventions, see section 20.3.2, page 466, of [8].[1] In these examples `a3` is an array of integer arrays, and `a4` is an array of object arrays.

Array Access

The elements of an array are accessed via:

[1]Java's internal name for an array class consists of the name of the element type (`B` for `byte`, `C` for `char`, `D` for `double`, `F` for `float`, `I` for `int`, `J` for `long`, `S` for `short`, `Z` for `boolean`, and `L`*classname*`;` for classes or interfaces) preceeded by one or more `[` characters, depending on the depth of array nesting. Thus `[[[[Z` would be the name for an array of arrays of arrays of arrays of booleans.

```
(aget <array> <exp>)
```

which returns the value of `<array>[<exp>]`, both `<array>` and `<exp>` are evaluated. For example, in the context of the previous array construction examples, both of these expressions

```
(aget a1 (int 0))
(aget (aget a3 (int 1)) (int 0))
```

evaluate to `88`. Whereas

```
(aget a2 (int 0))
(aget (aget a4 (int 1)) (int 0))
```

both evaluate to the same instance of the `java.io.PrintStream` class associated with the standard error stream.

Array Assignment

Similarly an array may be set via:

```
(aset <array> <exp> <expV>)
```

which both sets and returns the new value of `<array>[<exp>]`. All three subexpressions are evaluated from left to right. As in the case of array construction the value of the expression `<expV>` must be an allowable element of the array.

So to continue our running examples, we could modify our integer array, `a1`, by evaluating the expression

```
(aset a1 (int 0) (char 'Z'))
```

and accessing `a1` would reflect this change

```
(aget a1 (int 0))
```

and evaluate to `90`.

3.0.7 Strings

The Scheme reader will interpret any string of characters beginning and ending with the character " as the corresponding Java string (without the two occurrences of the character "), actually "foo" is interpreted as (quote foo), which evaluates to the Java String foo.

String concatenation is provided by the concatenation form:

```
(concat  <exp> .... <exp>)
```

This form of String concatenation, evaluates each expression and concatenates the result (using the toString() method of each object returned, and the usual conversions in the case of primitive data). So a simple example of this is that

```
(concat (char '\t') "A " "short " "sentence.")
```

evaluates to a string that prints as

```
        A short sentence.
```

3.0.8 Arbitrary Objects

Object Construction

Arbitrary Java objects may be constructed using the following form:

```
(object (<exp> <exp1> ... <expN>))
```

The arguments are, as usual evaluated from left to right, the first argument <exp> should evaluate to the full name of a Java class. If the so named Java class is not public an exception is thrown. The interpreter then attempts to find a constructor for that class with matching arguments, and using that constructor and arguments, constructs the appropriate object.

For example we can construct a generic object by evaluating

```
(object ("java.lang.Object"))
```

and a wrapper object of the character X via

```
(object ("java.lang.Character" (char 'X')))
```

A slightly more interesting example is a frame

```
(define frame (object ("java.awt.Frame" "A Frame")))
```

that we will futz with shortly.

However, first a little more needs to be said about the resolution or search for the appropriate constructor to use. It is, of course, the runtime types of the values of the arguments `<exp1>` ... `<expN>` that are used to determine the appropriate constructor. If an argument's value is `null` it's type is treated like a wild card.

N.B. Only constructors that are declared `public` qualify in this search.

The search proceeds as follows. First the interpreter looks for a constructor that exactly matches the argument types (using the

```
public Constructor getConstructor(Class[] parameterTypes)
```

method of the `java.lang.Class` class). Otherwise we look among the `public` constructors for the best match, the candidates are chosen from those returned by the

```
public Constructor[] getConstructors()
```

method of the `java.lang.Class` class, *not*, for example, the

```
public Constructor[] getDeclaredConstructors()
```

method of the `java.lang.Class` class). The notion of best is taken to mean the *least* when taking widening, the interface hierarchy, and the class hierarchy into consideration. If no constructor is found an exception is thrown.

The `null` object reference is obtained by the special form:

```
(object null)
```

which evaluates to the `null` object reference.

Field Access

Static and non-static fields of an object can be accessed via:

```
 (lookup <target> <field>)
```

The `<target>` and `<field>` positions are evaluated from left to right. The interpreter then looks for a field belonging to the object that the `<target>` evaluates to. The value of that field is returned. If no corresponding field is found an exception is thrown. If `<target>` does not evaluate to an object an exception is also thrown.

For example the following expressions both evaluate to

```
(lookup a1 "length")
(lookup a4 "length")
```

to the integer `3`.[2]

It is, of course, the runtime type of the value of `<target>` that is used in determining the appropriate field, and its value. (Since this is an interpreted language, there is no sensible notion of compile time type.) Thus `static` fields are probably best looked up via the class rather than an instance if there is the possibility of ambiguity.

The value of a static field may also be accessed from the class via the usual notation

```
java.lang.Math.PI
```

Method Invocation

Static and non static methods of an object may be invoked using the following form:

```
(invoke <target> <method> <exp1> ... <expN>)
```

The arguments are evaluated from left to right, and then the interpreter attempts to find a method, whose name is the value of `<method>` (which should be a `String`), with the appropriate arguments.

Thus for example we could configure and display our `frame` via the following sequence of invocations

```
(invoke frame "setSize"  (int 50) (int 50))
(invoke frame "setLocation" (int 10) (int 10))
(invoke frame "setVisible" (boolean true))
```

which would cause the `frame` to be a square of fifty pixels positioned in the top left hand corner of the screen.

Again more needs to be said about how the method gets selected. It is the runtime method of the corresponding value of `<target>` that is subsequently found and invoked. Unless that class is not `public`, in which we use the first `public` superclass. If no matching method is found an exception is thrown. The search is almost identical to the one undertaken in the object construction case. It is, of course, the runtime types of the values of

[2] That these evaluate correctly is due to an ad hoc clause in our interpreter, since from the point of view of Java's reflection API, `length` is not a field of an array, contradicting the Java language specification, see section 10.7, page 197, of [8]

the arguments `<exp1>` ... `<expN>` that are used to determine the appropriate method. As in the constructor case, if an argument's value is `null` it's type is treated like a wild card.

N.B. Only methods that are declared `public` qualify in this search.

First the interpreter looks for a method that exactly matches the argument types (using the

```
public Method getMethod(String name, Class[] parameterTypes)
```

method of the `java.lang.Class class)`. Otherwise we look among the `public` methods for the best match, the candidates are chosen from those returned by the

```
public Method[] getMethod()
```

method of the `java.lang.Class` class, *not*, for example, the

```
public Method[] getDeclaredMethods()
```

method of the `java.lang.Class` class). The notion of best is elaborated slightly from the constructor case to take into account the declared return types of the method. This is only used when the parameter types match exactly, and in this case the more specific return type is preferred. (See sections 8.2 and 8.4 of [8], and the API for the

```
public Method getMethod(String name, Class[] parameterTypes)
```

method of the `java.lang.Class` class). If no method is found an exception is thrown.

In the case where one wishes to invoke a `static` method via the class rather than the object we include the following form:

```
(sinvoke <target> <method> <exp1> ... <expN>)
```

In this case `<target>` should evaluate to a string that names the desired class. The evaluation then proceeds in a similar fashion to the non static case.

So for example if we wished to configure and position our `frame`, so that it was half the size of our screen, and centered therein, we could do the following

```
(define toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
(define dim (invoke toolkit "getScreenSize"))
(define h (lookup dim "height"))
(define w (lookup dim "width"))
(invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
(invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
(invoke frame "setVisible" (boolean true))
```

sequence of instructions.

To give another example, the class object corresponding to `java.lang.Math` can be obtained, and named, via the expression

```
(define math (sinvoke "java.lang.Class"  "forName" "java.lang.Math"))
```

But it would be a mistake to try and access static fields of the `java.lang.Math` class, say `PI`, by then doing

```
(lookup math "PI")
```

because this will actually end up looking for a field called `"PI"` in the class that `math` is an instance of, i.e. `java.lang.Class`. To access static fields via the class use:

```
java.lang.Math.PI
```

3.0.9 Control Forms

The usual Scheme/Lisp forms are provided.

Lexical Binding

Lexical binding is available via:

```
(let (
      (<var1> <exp1>)
      (<var2> <exp2>)
      ...
      (<varN> <expN>)
     )
 <exp>
 )
```

This expression incrementally augments the current lexical environment. I.e. the binding of `<var1>` to the value of `<exp1>` is visible in the evaluation of `<exp2>`, and all subsequent expressions. The value of the entire expression is the value of `<exp>` in the fully augmented environment. So for example if one were nostalgic for the initial UNIX process file descriptor table, one could construct the following array:

```
(let ((0 java.lang.System.in)
      (1 java.lang.System.out)
      (2 java.lang.System.err))
  (array java.lang.Object 0 1 2))
```

Sequencing

Sequencing is available via the form:

```
(seq <exp1> ... <expN>)
```

The value returned from a sequencing expression is the value returned by the last expression in the sequence.

Thus a simple use of these last two forms would be to construct our centered frame, without littering the global environment with debris.

```
(let ((frame (object ("java.awt.Frame" "A Frame")))
      (toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
      (dim (invoke toolkit "getScreenSize"))
      (h (lookup dim "height"))
      (w (lookup dim "width")))
  (seq
   (invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
   (invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
   (invoke frame "setVisible" (boolean true))
   frame)
)
```

Lambda Expressions and Closures

Closures are obtained by evaluating the corresponding lambda form:

```
(lambda (<param1> ... <paramN>) <exp>)
```

The value of such an expression is a closure, a pair consisting of the lambda expression, and the lexical environment at the time of evaluation. Thunks (the case when `N` is zero) are permissible. As an example of this we could define and name a centered frame factory as follows.

```
(define frameFactory
  (lambda (FrameName)
    (let ((frame (object ("java.awt.Frame" FrameName)))
          (toolkit (sinvoke "java.awt.Toolkit" "getDefaultToolkit"))
          (dim (invoke toolkit "getScreenSize"))
          (h (lookup dim "height"))
          (w (lookup dim "width")))
      (seq
        (invoke frame "setSize" (/ w (int 2)) (/ h (int 2)))
        (invoke frame "setLocation" (/ w (int 4)) (/ h (int 4)))
        (invoke frame "setVisible" (boolean true))
        frame)
    )
   )
)
```

Closure Application

Application of a closure to the appropriate number of arguments is via the apply form:

```
(apply <exp> <exp1> ... <expN>)
```

To litter our screen with annnoying popups we could then do

```
(apply frameFactory "Viagra popup")
(apply frameFactory "Nigerian bank scam")
(apply frameFactory "Lottery notification")
(apply frameFactory "Random porn site here!")
```

and save other people the time and effort.

Iteration

A iteration form is included based on the Scheme `do` form:

```
(do  (
       (<var_1> <init_1> <step_1>)
       ...
```

```
  (<var_N> <init_N> <step_N>)
 )
 (<test> <exp> ...)
 <command> ...)
```

Evaluation of a `do` form proceeds in two steps, an initialization phase followed by an iteration phase. In the initialization phase the `<init_i>` expressions are evaluated sequentially in the outer environment. Their values are then bound to the variables `<init_i>` to form the `do` environment, and the iteration phase commences. In each iteration the following takes place. The `<test>` expression is evaluated, if it does not evaluate to a `boolean` an exception is thrown. If it evaluates to `true`, then the `<exp> ...` are evaluated in order, and the value of the last expression is the value of the entire `do` form. Otherwise each of the `<command> ...` expressions are evaluated for their effect, then the variables in the `do` environment are updated, sequentially, to be the values of the `<step_i>` forms, and the next iteration commences.

So a simple example is

```
(do (
     (i (int 0) (+ i (int 1)))
     (str " " (concat str " " i))
    )
    ((= i (int 10))  str)
    (invoke java.lang.System.err "println" str)
    )
)
```

prints out a triangle of numbers:

```
 1
 1 2
 1 2 3
 1 2 3 4
 1 2 3 4 5
 1 2 3 4 5 6
 1 2 3 4 5 6 7
 1 2 3 4 5 6 7 8
 1 2 3 4 5 6 7 8 9
```

Branching

Both binary and ternary forms of the branching primitive are allowed:

```
(if <test> <then>)
(if <test> <then> <else>)
```

In both the binary and ternary forms the `<test>` should return a `boolean`, The binary form returns `null` if `<test>` is false.

Boolean Operations

The related propositional forms are as usual:

```
(and <exp> .... <exp>)
(or <exp> .... <exp>)
(not <exp>)
```

The expressions are evaluated from left to right, their values should be `boolean`, otherwise an exception is thrown. In the case of `and`, evaluation is terminated on the first `false` value, while in the case of `or`, evaluation is terminated upon the first `true` value encountered.

Quoting

Finally evaluation can be prevented via the usual quote form:

```
(quote <exp>)
```

which returns the unevaluated `<exp>` as data, either a `String` or a `List`.

3.0.10 Subtyping

There is a version of Java's `instanceof` operator. The expression:

```
(instanceof <exp> <exp>)
```

returns Java's idea of whether or not the value of the first `<exp>` is an instance of the class named by the (`String`) value of the second `<exp>`. E.g.

```
(instanceof "java.lang.String" "java.lang.String") ==> true
```

Note that there is a clear distinction made between primitive data and the associated wrapped form, an expression such as:

```
(instanceof (int 7) "java.lang.Integer")  ==> false.
```

3.0.11 Attributes

Attributes, akin to dynamic fields, are manipulated, for efficiency, via the dedicated forms:

```
(setAttr <exp> <exp> <exp>)
(getAttr <exp> <exp> [<default>])
```

rather than by the corresponding methods of the object involved (which must extend the `glyphics.Attributable` interface).

3.0.12 Miscellaneous Operations

A file may be loaded using the load expression:

```
(load <exp>)
```

Graphical objects that are manipulated by other actors (for example formal reasoning tools such as Maude and PVS) may be assigned unique identifiers by them. To access an object by it's unique identifier the fetch form is used:

```
(fetch <exp>)
```

An object, belonging to a class extending `glyphics.Identifiable`, may be assigned a unique identifier using the `setUID` method.

3.0.13 Class Names

All of Java's classes may be accessed by their full names. There is no import mechanism, other than that provided by the lexical and dynamic binding primitives:

```
(define Vector "java.util.Vector")


(let ((Line "java.awt.geom.Line2D$Double")
      (victor (object (Line (int 0) (int 0) length (int 0))))
  (object (Vector victor)))
```

Note that, following Java's reflection conventions, inner classes are accessed via the `$` rather than the `.`.

Chapter 4

A Complete `JLambda` Program

In this chapter we present a complete example of a `JLambda` program. The program, `clicker`, contains the most interesting features of `JLambda`, namely, the creation and manipulation of Java objects, and the use of closures [3] as event handlers. Furthermore, `clicker` is one of the benchmark programs used in Chapter 8 for comparing the execution efficiency of various versions of the `JLambda` interpreter.

When the `clicker` program is run, it displays a window containing a single circular node. Clicking anywhere in the window creates a new node at that point. The user can change the colour of an existing node by shift-clicking on it; a dialogue box will appear, allowing the user to select a new colour for the node. Finally, existing nodes can be dragged using the mouse.

The `clicker` program constructs graphical objects using the `glyphics` class hierarchy, which is a component of the IOP project. We provide an overview of the `glyphics` hierarchy in Section 4.1.

In Section 4.2 we present a "table of contents" for the clicker program code, making clear the lexical structure of the program to aid navigation. Section 4.3 contains details each section of the `clicker` program code, along with an explanation of its operation.

4.1 An Overview of the `glyphics` Hierarchy

The `glyphics` class hierarchy provides `JLambda` with enough built in classes to effectively construct, at runtime, any desired interactive graphical object.

The root class of all things glyph-like is the `abstract` class `Glyphish`. It has three main immediate concrete subclasses: the `Glyph`, the `GlyphList`, and the `ClosureGlyph`.

A `Glyph` instance has a single `java.awt.Shape`, border colour, fill colour, and stroke width. A `GlyphList` is a composite; it consists of an ordered list of `Glyphish` things. A `ClosureGlyph` is the most dynamic; it requires closures (in `JLambda`) to implement all the methods required by the `Glyphish` API.

A concrete `Glyphish` instance portrays itself by implementing the `abstract` method

```
public abstract void paint(java.awt.Graphics2D g2d);
```

declared in the `Glyphish` class. In the case of a `Glyph` instance it will draw itself according to its `java.awt.Shape` field, fill colour, border colour and stroke.

`Glyphish` instances are capable of handling any input events, that is, instances of the `InputEvent` class of the `java.awt.event` package. The `Glyphish` class implements each of the `Input` event listener interfaces: `MouseListener`, `MouseMotionListener`, and `KeyListener`, all of the package `java.awt.event`. They do so in a uniform way. For each method in the listener interface a `Glyphish` instance has a `Closure` object associated with it. For example, in the case of the

```
public void mouseClicked(java.awt.event.MouseEvent e);
```

method of the `MouseListener` class, the `Glyphish` class has the field

```
private glyphics.scheme.Closure mouseClickedAction;
```

Calling the

```
mouseClicked(java.awt.event.MouseEvent e)
```

method would result in the `clickedAction` closure being applied:

```
clickedAction.applyClosure(this, e);
```

where the `this` pointer is of the `Glyphish` instance that is responding to the `MouseEvent` instance `e` of the package `java.awt.event`.

Positioning, moving, and animating `Glyphish` instances is done by applying affine transformations (e.g. translating, rotating, shearing, and scaling) to them.

Finally, the two classes `glyphics.IOPFrame` and `glyphics.IOPView` together implement the top-level window for `glyphics` IOP applications.

4.2 Clicker Table of Contents

To aid navigation of the `clicker` source code, we provide the following representation of its lexical structure.

```
(let ((h ...)
      (w ...)
      (black ...)
      (yellow ...)
      (stroke ...)
      (ellipse ...)
      (view ...)
      (frame ...)
      (mkNode
       (lambda (...)
         (let ((node ...)
               (pressed ...)
               (released ...)
               (dragged ...)
               (clicked ...)
               (trans ...))
           (seq ... (invoke ...) ...))))
      (clickedV ...))
  (seq ...))
```

4.3 Code Listing for `clicker`

The `clicker` program begins by creating the main window object; it defines bindings for some Java primitive types, fields, and objects, which are used to specify the initial properties of the node object, and the `glyphics.IOPFrame` object, which is the program's main window.

```
(let ((h (int 50))
      (w (int 70))
      (black java.awt.Color.black)
```

```
      (yellow java.awt.Color.yellow)
      (stroke (object ("java.awt.BasicStroke" (float 2.5))))
      (ellipse (object ("java.awt.geom.Ellipse2D$Double"
                        (int 0) (int 0) w  h)))
      (view (object ("glyphics.IOPView"
                     (boolean true) (boolean true))))
      (frame (object ("glyphics.IOPFrame" "Node Example" view)))
```

Note that the use of these bindings relies on a particular semantics of `JLambda`'s `let` expression: that each binding incrementally augments the lexical environment.

The next code section defines a closure which, when invoked, constructs a node object.

```
(mkNode
 (lambda (xPos yPos)
   (let ((node (let ((temp (object ("glyphics.Glyph" ellipse black yellow))))
              (seq (invoke temp "setStroke" stroke)
                   temp)))
```

In creating the node, the closure uses several bindings from the previous section to specify initial properties, such as colour, of the node.

The `mkNode` closure also creates several closures for each of the mouse click events to which the node responds, and binds them to the variables `pressed`, `released`, `dragged`, and `clicked`. Later in the program, these closures will be registered as event handlers for the newly-created node.

The next section of the program contains the definition of each of these closures:

```
(pressed (lambda (self event)
           (seq
            (setAttr self
                     "pointF"
                     (object ("java.awt.geom.Point2D$Double"
                              (invoke event "getX")
                              (invoke event "getY"))))
            (setAttr self "draggedF" (boolean true)))))

(released (lambda (self event)
```

```
            (setAttr self "draggedF" (boolean false))))

(dragged (lambda (self event)
           (if (getAttr self "draggedF")
               (let ((pnt (getAttr self "pointF"))
                     (eX (invoke event "getX"))
                     (eY (invoke event "getY"))
                     (a (let ((temp (object "java.awt.geom.AffineTransform")))
                          (seq
                           (invoke temp
                                   "translate"
                                   (- eX (invoke pnt "getX"))
                                   (- eY (invoke pnt "getY")))
                           temp))))
                 (seq (invoke self "transform" a)
                      (invoke pnt
                              "setLocation"
                              (invoke event "getX")
                              (invoke event "getY"))
                      (invoke view "repaint"))))))

(clicked (lambda (self event)
           (if (invoke event "isShiftDown")
               (let ((chooser (object ("javax.swing.JColorChooser")))
                     (color (invoke chooser
                                    "showDialog"
                                    frame
                                    "Color Chooser"
                                    (invoke self "getFill"))))
                 (seq (if (!= color (object null))
                          (invoke self "setFill" color))
                      (invoke view "repaint"))))))
```

When an event handler closure is invoked it receives two arguments: the object receiving the event, and the event object itself. The closure uses information stored in the event object to manipulate the receiving object.

Each of the above closures manipulate the node object in ways that correspond to the type of event the node object received. For example, the `clicked` closure checks whether the shift key is depressed; if the key is depressed, the closure displays a dialogue box, and changes the colour of the node to the selected colour.

Next, a `java.awt.geom.AffineTransform` object is defined, which will later be used to provide translation of the node object in response to a mouse drag event.

```
(trans (let ((temp (object ("java.awt.geom.AffineTransform"))))
         (seq (invoke temp "translate" xPos yPos)
              temp))))
```

The event handling closures that have been created and bound to variables must now be registered as event handlers for the node object. The following code sequence invokes the `setMouseAction` method of the node object to register closures for each of the different types of mouse event to which the node responds.

```
(seq
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_PRESSED
         pressed)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_RELEASED
         released)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_CLICKED
         clicked)
 (invoke node
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_DRAGGED
         dragged)
 (invoke view "add" node trans)))))
```

The final `invoke` expression in the preceding code sequence simply adds the translated node to the view object.

The program now defines a closure that creates a new node in response to a mouse click event.

```
(clickedV (lambda (self event)
            (if (not (invoke event "isShiftDown"))
                (seq
                 (apply mkNode
                        (- (invoke event "getX") (/ w (int 2)))
                        (- (invoke event "getY") (/ h (int 2))))
                 (invoke view "repaint"))))))
```

The `clickedV` closure receives the window object and event object as arguments, and uses the event object to determine the position at which to create the new node. `clickedV` creates a new node object by calling the `mkNode` closure defined previously.

Since a mouse click event creates a new node, it must be received by the window object, and not a node object. Consequently, the closure that handles mouse click events is not created within the body of `mkNode`, unlike the previously defined event handler closures.

The remaining section of code simply registers the node-creation event handler, `clickedV`, with the window object, and displays an initial node.

```
(seq
 (invoke view
         "setMouseAction"
         java.awt.event.MouseEvent.MOUSE_CLICKED
         clickedV)
 (apply mkNode (* w (int 3)) (* h (int 3)))
 (invoke view "repaint")))
```

After the program has been evaluated the interpreter exits, and Java's AWT thread continues running to listen for window events. Closures that are invoked in response to events are interpreted in this second thread.

The code listing for the `clicker` program demonstrates the power and simplicity of the interface provided the `JLambda` language to Java's language facilities. Features such as closures and lexical scoping make construction of event-driven graphical programs, such as `clicker`, particularly straight-forward.

Chapter 5

A Recursive Interpreter for `JLambda`

We present in this chapter our first-pass design of an interpreter for the `JLambda` language. The goal of the design is merely a simple working interpreter; we are not yet concerned with the interpreter's performance. Our design is a simple recursive evaluator similar to that commonly presented in undergraduate courses on programming language implementation [1]. We explain in the following sections the major components of the design—the parser, the environments, and the evaluation model. Although this design has the virtue of a simple implementation, it has a major flaw when the implementation language is Java. We detail this flaw in the final section.

5.1 The `List` Class

We begin the explanation of the interpreter's operation by presenting the `List` class. The `List` class is used heavily by the interpreter—expressions and environments are represented using `List`s. Since there will be references to the `Lisp` class throughout the rest of this chapter, we explain here operations it provides.

In keeping with Lisp tradition, we call the first item in a list the *car* of the list; the rest of the list—that is, the part of the list that follows the car—is called the *cdr* of the list.

The `List` class is an implementation of the Lisp list, with one anomaly: our implementation of the empty `List` is actually a cons cell with the empty flag set. We follow Lisp tradition and denote the head and tail of a list by `car` and `cdr`, respectively. The elements of a `List` can be any Java `Object`. The `List` class provides all of the expected operations, such as list construction, `car` and`cdr` accessors, and so on.

The following `List` constructors are provided to create new empty lists

```
public List();
```

or new lists with the specified `car` and `cdr`.

```
public List(Object car, List cdr);
```

The test for whether a `List` is empty is provided by the `isEmpty` method.

```
public boolean isEmpty();
```

The `List` class provides static convenience methods for the creation of zero-element, singleton, two-element, and three-element `List`s.

```
public static List list();
public static List list(Object o);
public static List list(Object o1, Object o2);
public static List list(Object o1, Object o2, Object o3);
```

The accessor operations, `car` and `cdr` have the usual behaviour.

```
public final Object car();
public final List cdr();
```

The `List` class also provides the convenience accessor methods `caar`, `cadr`, `cdar`, `cddr`, and `caddr`, for performing the composition of `car` and `cdr` operations. The names consist of a `c`, followed by a series of `a`'s or `d`'s, and finally an `r`. The series of `a`'s or `d`'s is chosen to identify the series of `car` and `cdr` operations that is performed by the method. The order in which the `a`'s and `d`'s appear is the inverse of the order in which the corresponding operations are performed.

For selecting an arbitrary element of a list, the `List` class is equipped with the operation `nth`.

```
public Object nth(int n);
```

Finally, the size of a `List` object can be obtained by using the `size` operation.

```
public int size();
```

In addition to the methods already presented, the `List` class contains several methods related to parsing of interpreter input. We present those methods in Section 5.2.

5.2 The Parser

The first stage of evaluation of a `JLambda` expression is the parsing of the input. The input may be read from a file or taken from the interactive command interpreter. If the input is read from a file, all of the forms contained in the file are read until the end-of-file marker is encountered. If the input is taken from the command interpreter, only a single form is read.

The parser consumes the input and returns, for each form present in the input, either a single token or an S-expression whose atoms are tokens. The tokenising of the input is handled by the Java `StringTokenizer` class. A token returned by the parser is simply a Java `String`.

The parser merely parses S-expressions (as a character stream) and returns S-expressions; it does not compile input into an abstract expression representation.

5.3 Environments

We use two kinds of environments in the interpreter, a global environment and a lexical environment. A global environment is responsible for maintaining the global definitions introduced by the `define` form. We implement the global environment as a hash table that maps name to values.

Our design for lexical environments is simple: a lexical environment is a list of variable-value pairs, and environment lookups use linear search.

5.4 The Evaluation Model

The evaluation model used by the interpreter is a simple recursive evaluation procedure. Evaluation is structured as a case analysis of the syntactic type of the expression to be evaluated, and begins when a form is received from the parser. The evaluator determines the type of the form by examining the token at the head of the form, which we call the *form ID*. The evaluator then dispatches on the form ID to a method responsible for the evaluation of forms of the type specified by the form ID. This process is then repeated recursively for each subexpression.

The following two rules describe the essence of the recursive evaluation model:

1. To evaluate closure application, evaluate the closure subexpression and then apply the value of the closure subexpression to the values of the argument subexpressions.

2. To apply a closure to a set of arguments, evaluate the body of the closure in a new environment. To construct this environment, extend the environment part of the closure with bindings for the formal parameters to the arguments.

An example will serve to elucidate this process. Consider the evaluation of the following `apply` expression:

```
(apply (lambda (x) (+ x x)) (int 4))
```

The core of the evaluator is a set of overloaded static `evaluate` methods of the `Evaluate` class. The entry point of the evaluator is the following `evaluate` method, which receives from the parser a `List` that represents the parsed expression:

```
public static Object evaluate(List expr) {
    return evaluate(expr, new Environment());
}
```

The `evaluate` method simply passes the expression it receives to an `evaluate` method that takes an additional argument, the environment in which the expression is to be evaluated. Since the `evaluate` method defined above receives only top-level expressions, it passes on an empty environment.

The following is the definition of the second `evaluate` method. We include only the parts that are relevant to the evaluation of `apply` expressions:

```
public static Object evaluate(Object form, Environment env) {
    ...
    if (form instanceof List) {
        String tag = form.car();
        ...
        if (tag.equals("apply")) {
            return evaluateApply(form.cdr(), env);
        }
        ...
    }
}
```

This `evaluate` method handles the evaluation of all expressions, and is structured as a case analysis of the syntactic type of the expression to be evaluated. In our example, `evaluate` dispatches on the expression tag (`"apply"`) to the `evaluateApply` method, which is responsible for evaluating `apply` expressions. The `evaluate` method passes to the `evaluateApply` method the body of the `apply` expression—the `lambda` expression and arguments—and also passes along the environment it received.

Here is the definition of `evaluateApply`:

```
public static Object evaluateApply(List body, Environment env)
    throws EvaluateError {
    int len = body.size();
    if (len == 0) {
        throw new EvaluateError("body wrong length: " + len);
    }

    Object funObj = evaluate(body.car(), env);
    if (!(funObj instanceof Closure)) {
        throw new EvaluateError("function not a closure" + funObj);
    }
    Closure closure = (Closure) funObj;

    List        closureParams = closure.getParams();
    Object      closureBody   = closure.getBody();
    Environment closureEnv    = closure.getEnv();

    List args = body.cdr();
    if (parameters.size() != args.size()) {
        throw new EvaluateError("closure params and args differ"
                                + " in length");
    }

    Environment envApply = new Environment(closureParams,
                                           evaluateList(args, env),
                                           closureEnv);
    return evaluate(closureBody, envApply);
```

```
}
```

The `evaluateApply` method makes a recursive call to `evaluate` to evaluate the `lambda` expression, and is returned a `Closure`. `evaluateApply` then calls `evaluateList` to evaluate the application arguments. `evaluateList` makes recursive calls to `evaluate` to evaluate each expression in the list it receives. In this case, `evaluateList` returns a `List` containing the argument value, `4`.

Finally, the body of the closure is evaluated—again by recursively calling `evaluate`—in an environment obtained by extending the environment stored in the closure with a binding for the formal parameter and argument value. The value returned by this call to `evaluate` is the value of the complete `apply` expression; this value is the return value of the evaluator.

This example highlights the key to the operation of the interpreter: the evaluation of the subexpressions of the `apply` expression—the `lambda` expression and the argument expressions—as well as the evaluation of the `apply` expression itself, is carried out by recursive calls to `evaluate`.

Although the interpreter correctly evaluates expressions such as our example, the recursive nature of its evaluation model, combined with the lack of proper tail recursion [5] in the Java virtual machine, causes it to fail on certain other kinds of expressions.

5.5 The Flaw in the Design

We explain the problem with the current design of the interpreter with another example, a program that implements an infinite loop. Clearly, this is not a realistic program; it does however providen a simple demonstration of the problem with the current interpreter implementation.

```
(seq
  (define inf-loop () (apply inf-loop))
  (apply inf-loop))
```

We would of course like our interpreter to evaluate this expression—the interpreter should execute an infinite loop. Instead, when we evaluate the above expression we discover that the interpreter fails with a stack overflow error.

To see why this happens, we again trace the evaluation of the expression. The interpreter evaluates the `apply` expression by making a recursive call to `evaluate`, and passes to `evaluate` the body of the `apply` expression, which is another identical `apply` expression. The interpreter repeats this process indefinitely. Since the Java virtual machine handles method invocation by pushing a stack frame, the stack eventually overflows.

This example demonstrates that the interpreter is unable to handle arbitrarily long recursions, and therefore cannot evaluate many real-world programs, such as those that compute long lists.

Our first-pass design for the interpreter is flawed because it uses a recursive process, which eventually causes a stack overflow error. In the next chapter we show how this flaw can be removed by changing the design of the interpreter to use an iterative process.

Chapter 6

A Register Machine Interpreter for `JLambda`

We saw in Chapter 5 that the interpreter causes a stack overflow error when it attempts to evaluate certain expressions. The source of this problem is that, because it inherits the control structure of the underlying Java system, the interpreter executes as a recursive process. In this chapter we demonstrate the changes to the interpreter design necessary to implement the interpreter as a register machine. In the new design, the procedure-calling and argument-passing mechanisms used in the evaluation process can be described in terms of operations on registers. We thus obtain an explicit-control evaluator, which exhibits iterative execution behaviour.

The transformation of the recursive interpreter into one with iterative behaviour involves two steps. First, we ensure all recursive calls are tail calls. We achieve this by transforming the interpreter into continuation passing style [12]. Such a transformation would be sufficient to achieve an interpreter with iterative execution if our interpreter was implemented in a properly tail recursive language. In properly tail recursive languages tail recursion is guaranteed to be equivalent to iteration. Since our implementation language is Java—which is not tail recursive [5]—we must perform a further step to manually transform tail recursion into iteration.

The second step, therefore, is the transformation of the interpreter from a continuation passing style into a register-based imperative style. This transformation is based on the following observation: if a set of methods call each other only by tail calls, we can first rewrite the calls to use variable assignment instead of argument-passing, and we can then replace method calls with jumps. The register machine transformation consists of systematically performing such rewrites.

In this thesis we do not discuss the continuation passing style transformation or the register machine transformation. Detailed discussions of both transformations can be found in Matthias Felleisen's Ph.D. dissertation[6].

Once these two steps are complete we will have arrived at an interpreter with iterative execution behaviour. The interpreter resulting from the first step—a continuation passing style interpreter—will not be presented; we focus in this chapter on the interpreter obtained by performing both steps. This register machine interpreter can evaluate any `JLambda` expression without overflowing the Java virtual machine stack.

6.1 The Register Machine

Our `JLambda` evaluator register machine includes four registers: `exp`, `val`, `env`, and `k`. `Exp` is used to hold the expression to be evaluated, and `env` contains the environment in which the evaluation is to be performed. At the end of an evaluation, `val` contains the value obtained by evaluating the expression in the designated environment. The `k` register is used to implement recursion, and stores the continuation to be invoked upon completion of an evaluation.

The registers form part of the `State` class, and are implemented as fields of `State`.

```
class State {
    int          mode;

    Object       exp;
    Environment  env;
    Object       val;
    Continuation k;
    ...
}
```

The `State` class encapsulates the state of the register machine evaluator. The `mode` field represents the current mode of operation of the evaluator. There are four possible modes of operation: `EVAL`, `CONTINUE`, `RETURN`, and `DONE`. The evaluator is in `EVAL` mode when it begins evaluation of the expression in the `exp` register. The `CONTINUE` mode corresponds to the case when an expression is partially evaluated and control has been transferred to a continuation to complete the evaluation. The evaluator is in `RETURN` mode when evaluation of the expression contained in the `exp` register is complete, and its value has been placed in the `val` register. The `DONE` mode signifies that no more expressions remain to be evaluated; the value in the `val` register is returned by the interpreter as the value of the overall expression.

The core of the register machine evaluator is the `step` method of the `State` class. When the interpreter executes `step` it performs a single step of the evaluation process. The interpreter evaluates a top-level expression by first initialising the `State` object and then entering a loop, in which it repeatedly invokes the `step` method until the `State` object's mode of operation becomes `DONE`. Upon termination of the loop, the interpreter returns the contents of the `val` register as the value of the expression.

```
public static Object evaluate(Object form, Environment env) {
    State state = new State();
    state.init(form, env);

    while (state.mode != DONE) {
        state.step();
    }

    return state.val;
}
```

The `step` method of the `State` class co-ordinates expression evaluation based on the evaluator's current operation mode. The `EVAL` case corresponds to the `evaluate` method of the recursive evaluator described in Chapter 5. When the evaluator is in the `EVAL` mode it evaluates the expression specified by `exp` in the environment specified by `env`. When the evaluation is complete, the interpreter will be in the `RETURN` mode; it will invoke the continuation stored in `k`, and the `val` register will hold the value of the current expression. As with the recursive `evaluate`, the structure of the `EVAL` case of the `step` method is a case analysis on the syntactic type of the expression to be evaluated.

```
public void step() {
```

```
    switch (mode) {
    case EVAL:
        ...
        String formId = exp.car();

        if (formId.equals("=")) {
            ...
        } else if (formId.equals("and" ) {
            ...
        } else if (formId.equals("apply") {
            ...
        } else if ...
        ...
        break;

    case CONTINUE:
        ...

    case RETURN:
        k.ret(this);
        break;
    }
}
```

6.2 Continuations

We represent continuations by Java classes. A continuation can be viewed as corresponding to the stack in an executing program. With this in mind, we implement a chain of continuations as a kind of linked list: each continuation is an object that contains a reference to another continuation object, with the chain terminated by a special top-level continuation. The `Continuation` class is an abstract class that is the superclass of all continuation objects. For each of the different kinds of expressions understood by the interpreter, we define a corresponding subclass of `Continuation`. We also define a continuation subclass, `TopCont`, that will terminate the chain.

We define continuation classes for all `JLambda` expression types except variables, primitive data expressions, quotations, and `lambda` expressions, and these classes are responsible for co-ordinating the evaluation of that expression type. All of the continuation classes extend the abstract `Continuation` class. Continuation classes contain six fields: `formId`, `args`, `vals`, `n`, `env`, and `k`. The `formId` field contains the identification tag of the expression currently being evaluated. The `args` and `vals` fields contain the unevaluated and evaluated arguments, respectively. The number of arguments that have been evaluated so far is stored in the `n` field. The `env` field contains the environment in which the current expression is being evaluated. The `k` field contains the continuation to be invoked once evaluation of the current expression is complete.

Continuation classes implement three methods: `init`, `cont`, and `ret`. The `init` method initialises the fields of the `Continuation`. The `cont` method executes the next step of evaluation of an expression; it is called when the `Continuation` is initially invoked, and also after a subexpression has been evaluated. The `ret` method is called when the evaluation of the current expression is complete; it updates the operation mode of `State` and invokes the `Continuation` stored in `k`.

```
abstract class Continuation {
    String       formId;
    List         args;
    ArrayList    vals;
    int          n;
    Environment  env;
    Continuation k;

    public void init(String       f,
                     List         a,
                     Environment  e,
                     Continuation aK) {
        formId = f;
        args   = a;
        vals   = new ArrayList();
        n      = 0;
        env    = e;
        k      = aK;
```

```
    }

    abstract void cont(State s);
    abstract void ret(State s);
}
```

The chain of continuations is terminated by a top level continuation, whose only purpose is to terminate the evaluation loop, which it achieves by setting the interpreter's operation mode to `DONE`. We implement this continuation with the `TopCont` class.

```
class TopCont extends Continuation {
    public TopCont() {
    }

    public void cont(State state) {
        throw new UnsupportedOperationException();
    }

    public void ret(State state) {
        state.tag = DONE;
    }
}
```

The `TopCont` continuation is created when evaluation of a top level expression begins, during the initialisation of the `State` object.

```
class State {
    ...

    public void init(Object x, Environment n) {
        exp  = x;
        env  = n;
        k    = new TopCont();
        mode = EVAL;
    }
}
```

6.3 The Evaluation Process

Variables, primitive data expressions, quotations, and `lambda` expressions have no subexpressions to be evaluated. For these, the evaluator simply places the appropriate value in the `val` register and continues execution by invoking the `ret` method of the continuation stored in the `k` register. Evaluation of such expressions is performed by the following parts of the `step` method of the `State` class:

```
public void step() {
    switch (mode) {
    case EVAL:
        if (exp instanceof String) {
            val  = lookupVariable(exp, env);
            mode = RETURN;
            break;
        }

        if (!(exp instanceof List)) {
            val  = exp;
            mode = RETURN;
            break;
        }

        List form = (List) exp;
        if (form.size() < 2) {
            throw new EvaluateError("list too short -- " + form);
        }

        Object carObj = form.car();
        if (!(carObj instanceof String)) {
            throw new EvaluateError("form tag not a String -- "
                                    + carObj);
        }

        String formId = (String) carObj;
        if (formId.length() < 1) {
```

```
            throw new EvaluateError("formId too short -- " + formId);
        }

        List body = form.cdr();

        ...

        if (formId.equals("boolean")) {
            val  = Boolean.valueOf(body.car());
            mode = RETURN;
        } else if (formId.equals("byte")) {
            ...
        }
        ...
        } else if (formId.equals("lambda")) {
            int len = body.size();
            if (len != 2) {
                throw new EvaluateError("body wrong length -- "
                                         + len);
            }

            Object paramsObj = body.car();
            if (!(paramsObj instanceof List)) {
                throw new EvaluateError("params not a List -- "
                                        + paramsObj);
            }

            val = new Closure((List) paramsObj, body.cadr(), env);
            mode = RETURN;
        }
        ...

    case RETURN:
        k.ret(this);
        break;
```

```
    }
}
```

6.4 An Example of Evaluation

To illustrate the evaluation process for more complicated expressions we again use an `apply` expression as an example.

```
(apply (lambda (x) (+ x x)) (int 4))
```

The operator of an `apply` expression is a subexpression whose value should be a closure, and the operands are subexpressions whose values are the arguments to which the closure should be applied. We saw in Chapter 5 that the recursive evaluator handles `apply` expressions by calling itself recursively to evaluate each subexpression, and then calling itself to evaluate the body of the closure in an extended environment. The register machine evaluator does essentially the same thing, except that the recursive calls are implemented by invocations of `Continuations`. The invoked `Continuations` save the registers that will later be restored when the recursive call returns.

Evaluation begins at the `evaluate` method of the `Evaluate` class, as it did in the recursive evaluator of Chapter 5. This time, however, `evaluate` method simply steps the evaluation process by repeatedly invoking the `step` method of `State` until the mode of `State` becomes `DONE`. We first initialise the registers of the `State` object: we store the `apply` expression in the `exp` register; we store the empty environment (which is received as an argument) in the `env` register; and we create a `TopCont` continuation and store it in the `k` register. We then set the operation mode to `EVAL`.

We begin the evaluation of an `apply` expression by evaluating the operator to produce a closure, which will later be applied to the evaluated operands. This task is handled by the `ApplyCont` continuation. the first part of evaluation, therefore, is to create an `ApplyCont` continuation and store it in the `k` register. We pass to the `ApplyCont` constructor the current contents of the `k` register, which in this case is a `TopCont` continuation. Once the `ApplyCont` continuation has evaluated the `apply` expression, it will invoke the saved `TopCont` continuation to terminate the evaluation process.

The `env` register already contains the correct environment in which to evaluate the operator. We save `env`, however, because we will need it later to evaluate the operands.

For convenience, we also save, separately, the form identifier and the form body. The mode of the `State` object is then set to `CONTINUE`, and the `cont` method of the `ApplyCont` continuation is invoked.

```
public void step() {
    switch (mode) {
    case EVAL:
        ...
        if (formId.equals("apply")) {
            k = new ApplyCont(formId, body, env, k);
            mode = CONTINUE;
        }
        ...
        break;

    case CONTINUE:
        k.cont(this);
        break;

    ...
    }
}
```

To evaluate the operator—the `lambda` expression—we move it to the `exp` register and set the mode of `State` to `EVAL`.

```
class ApplyCont extends Continuation {
    public ApplyCont(String       exp,
                     List         args,
                     Environment  env,
                     Continuation k) {
        init(exp, args, env, k);
    }

    public void cont(State state) {
        state.exp  = args.nth(n);
```

```
        state.mode = EVAL;
    }
    ...
}
```

When the `cont` method returns, the `step` method is again invoked with the `exp` register now containing the `lambda` expression. The `lambda` expression is evaluated according to the process already described. The corresponding closure is then created and placed in the `val` register, and the mode is set to `RETURN`. The `k` register has not been changed during these operations, and still contains the `ApplyCont` continuation, since we want execution to return there to continue the evaluation of the remaining subexpressions of the `apply` expression. The `step` method invokes the `ret` method of `ApplyCont`.

Upon return from the evaluation of the operator subexpression, we proceed to evaluate the operands of the `apply` expression, and to accumulate the resulting arguments in a list, which is held in the `args` field of the `ApplyCont` continuation. If any operands still remain to be evaluated, we again set the mode to `CONTINUE` and repeat the loop, evaluating each operand in turn, accumulating its value in the `vals` field.

Each cycle of the argument-evaluation loop evaluates a single operand from the list stored in the `args` field of the `ApplyCont` object, and accumulates the result into the `vals` field. To evaluate an operand, we place it in the `exp` register and set the mode to `EVAL`, after setting `k` to the currently-executing `ApplyCont` object, so that execution will resume with the argument-accumulation phase.

We add the closure stored in the `val` register to `vals`, the list of evaluated arguments, and increment `n`, the number of evaluated arguments. Since the second argument (the operand, `(int 4)`) is still to be evaluated, we set the mode to `CONTINUE`, and then return.

```
class ApplyCont extends Continuation {
    ...
    public void ret(State state) {
        vals.add(state.val);
        n++;
        if (n < args.size()) {
            state.mode = CONTINUE;
        } else {
            ...
```

```
        }
    }
}
```

Evaluation of primitive data expressions such as `(int 4)` requires simply creating a corresponding primitive data object and storing it in the `val` register. The mode is set to `RETURN`, and the `step` method again invokes the `ret` method of the `ApplyCont` continuation object.

The `ret` method of `ApplyCont` stores the newly created primitive data object in the `vals` list, and increments `n`.

Once all of the operands have been evaluated, we perform the closure application. By this time the `vals` field of the `ApplyCont` object contains the closure to be applied, along with the evaluated arguments to which it must be applied. The `k` field contains the saved continuation which tells where to return with the result of the application. When the application is complete, the evaluator invokes the `ret` method of the saved continuation, with the result of the application in the `val` register.

```
class ApplyCont extends Continuation {
    ...
    public void ret(State state) throws EvaluateError {
        n++;
        if (n == 1 && !(state.val instanceof Closure)) {
            throw new EvaluateError("function not a closure -- "
                                    + state.val);
        }

        vals.add(state.val);

        if (n < args.size()) {
            ...
        } else {
            Closure closure = (closure) state.val;

            List        parameters = closure.getParams();
            Object      body       = closure.getBody();
```

```
            Environment closureEnv = closure.getEnv();

            Environment envApply = new Environment(parameters,
                                                   vals.cdr(),
                                                   closureEnv);


            state.exp  = body;
            state.env  = envApply;
            state.k    = k;
            state.mode = EVAL;
        }
    }
}
```

To apply a closure, we proceed just as with the recursive evaluation model of Chapter 5. We extend the environment contained in the closure with bindings for the closure's parameters and arguments; we then evaluate in this extended environment the expression that forms the body of the closure. Importantly, the continuation of the evaluation of the closure body is now the continuation of the `apply` expression—this is the key to the implementation of tail recursion.

The `ret` method of the `ApplyCont` class sets the registers in preparation for the evaluation of the closure body in the augmented environment, and control returns to the `step` method. The `exp` register contains the body of the closure, `(+ x x)`; the `env` register contains the environment that was stored in the closure, augmented with the binding for the variable `x`; and the `k` register now contains the `TopCont` continuation object.

The interpreter repeats this procedure to evaluate the `(+ x x)` expression, and the result is stored in the `val` register. The `ret` method of `TopCont` is now invoked, which sets the operation mode to `DONE`. The evaluation then terminates, and the sum is returned by the interpreter as the value of the complete `apply` expression.

6.5 Evaluation of `let` Expressions

The example presented in Section 6.4 demonstrates how evaluation proceeds for expressions in which each subexpression must be evaluated. However, to evaluate expressions such as if, let, and do we cannot simply evaluate all of their subexpressions. Evaluation of these expressions is more complicated. In this section we explain the process of evaluation of a let expression.

As usual, the evaluation process begins with the creation of an appropriate continuation object, in this case a LetCont object. The LetCont class uses the args field to store the list of bindings and the vals field to store the value of each binding expression; the LetCont class therefore defines an extra field, body, that stores the body of the let expression.

This is the LetCont constructor:

```
class LetCont extends Continuation {
    private Object body;

    public LetCont(String aFormId,
                   List someArgs,
                   Environment anEnv,
                   Continuation aK) {
        init(aFormId, (List) someArgs.car(), anEnv, aK);
        body = someArgs.cadr();
    }
}
```

Once the LetCont object is created, State.step invokes the LetCont object's cont method, which initiates evaluation of the binding expressions. cont evaluates each binding expression in the binding list by placing it in the exp register, then setting the evaluator mode to EVAL, and finally returning. The k register still stores the currently-executing LetCont object, and is left unchanged so that the step method calls LetCont's ret method, which retrieves the value of the binding expression and organises for the next binding expression to be evaluated.

```
void cont(State state) throws EvaluateError {
    if (n < args.size()) {
        Object binding = args.nth(n);

        if (!(binding instanceof List)) {
            throw new EvaluateError("let: bindings not an alist -- "
```

```
                                    + args);
        }

        int len = ((List) binding).size();
        if (len != 2) {
            throw new EvaluateError("let: binding not a pair -- "
                                    + binding);
        }

        Object key = ((List) binding).car();
        if (!(key instanceof String)) {
            throw new EvaluateError("let: key not a String -- " + key);
        }

        state.exp = ((List) binding).cadr();
    } else {
        state.exp = body;
        state.k = k;
    }

    state.tag = State.EVAL;
    state.env = env;
}
```

After each binding expression is evaluated the `ret` method of `LetCont` is called. The `ret` method adds the value of the binding expression, which is stored in the `val` register, to the `args` field. It then updates the environment stored in the `env` field with a binding from value it has just received to the corresponding variable, which is retrieved from the list of bindings stored in the `args` field. The evaluator mode is then set to `CONTINUE` so that the `cont` method is called by `State.step`, and continues the evaluation of the `let` expression.

```
void ret(State state) throws EvaluateError {
    String var = (String) ((List) args.nth(n)).car();

    env = env.extend(var, state.val);
```

```
    n++;
    state.tag = State.CONTINUE;
}
```

Finally, when all of the binding expressions have been evaluated and the environment stored in the `env` field has been extended with appropriate bindings, the body of the `let` expression is vevaluated. The `cont` method places in the `exp` register the expression stored in the `body` field; sets the `k` register to the continuation of the `let` expression; sets the evaluator mode to `CONTINUE`; and then returns.

6.6 Execution Behaviour of the Register Machine Interpreter

To show that the implementation of the interpreter as a register machine has iterative execution behaviour, we trace the execution of the infinite loop expression of Chapter 5. We reproduce here the code for the infinite loop:

```
(seq
  (define inf-loop () (apply inf-loop))
  (apply inf-loop))
```

Evaluation begins with the following section of the `State` class's `step` method. `step` creates a `SeqCont` continuation object, and invokes its `cont` method.

```
public void step() {
    switch (mode) {
    case EVAL:
        ...
        if (formId.equals("seq")) {
            k = new SeqCont(formId, body, env, k);
            mode = CONTINUE;
        }
        ...
        break;
```

```
    case CONTINUE:
        k.cont(this);
        break;

    ...
    }
}
```

The `cont` method of `SeqCont` stores the `define` expression in the `exp` register, and sets the operation mode of the `State` object to `EVAL`.

```
class SeqCont extends Continuation {
    ...

    public void cont(State state) {
        state.exp  = args.nth(n);
        state.env  = env;
        state.mode = EVAL;
    }
}
```

Execution now returns to `step`, which sets up evaluation of the `define` expression. A `DefineCont` continuation is created and its `cont` method invoked.

```
public void step() {
    switch (mode) {
    case EVAL:
        ...
        if (formId.equals("define")) {
            k = new DefineCont(formId, body, env, k);
            mode = CONTINUE;
        }
        ...
        break;

    case CONTINUE:
```

```
        k.cont(this);
        break;

    ...
    }
}
```

The `DefineCont` continuation object evaluates `define` expressions by first checking whether the value being defined is a simple value or a closure. It does this by checking the number of arguments in the `define` expression; if there are 3 arguments, it evaluates the `lambda` expression in the current environment, and adds the definition to the global environment. The defined name is placed in the `val` register, and returned as the value of the `define` expression. The saved continuation object, (a `SeqCont` object) is placed in the `k` register. Finally, the operation mode is set to `RETURN`.

```
class DefineCont extends Continuation {
    ...

    public void cont(State state) {
        Object name = args.car();

        if (args.size() == 2) {
            ...
        } else {
            Closure c = Evaluate.evaluateLambda(args.cdr(), env);
            Evaluate.addDefinition(name, c);

            state.val  = name;
            state.k    = k;
            state.mode = RETURN;
        }
    }

    ...
}
```

The `step` method then invokes the `ret` method of the `SeqCont` object.

The `ret` method of the `SeqCont` object stores the contents of the `val` register—the `inf-loop` variable—in its `vals` field. The second argument of the `seq` expression—the `apply` expression—still remains to be evaluated; `ret` sets the operation mode to `CONTINUE`, and execution returns to the `step` method. The `k` register is left unchanged; it still contains the `SeqCont` continuation object.

```
class SeqCont extends Continuation {
    ...

    public void ret(State state) {
        n++;
        vals.add(state.val);

        if (n < args.size()) {
            state.mode = CONTINUE;
        } else {
            ...
        }
    }

}
```

Control now returns to `step`, which immediately invokes the `cont` method of `SeqCont`. `cont` sets up the evaluation of the second argument of the `seq` expression, by first placing the `apply` expression in the `exp` register, then placing the saved environment in the `env` register, and finally setting the mode to `EVAL`.

The `step` method evaluates the `apply` expression by creating an `ApplyCont` continuation and invoking its `cont` method.

The `cont` method of `ApplyCont` sets up the registers for evaluation of its first argument, the variable `inf-loop`.

Execution returns to the `step` method, which evaluates the variable `inf-loop`. A lookup of the global environment is performed, and the closure bound to `inf-loop` is placed in the `val` register. `step` then sets the operation mode to `RETURN`, and invokes the ret method of the continuation object stored in the k register, which is still an ApplyCont object.

The cont method of the ApplyCont object applies the closure by evaluating the closure's body. Since the body of the closure is an apply expression identical to the one currently being evaluated, the evaluator loops indefinitely.

Unlike with the recursive interpreter implementation, the execution of the infinite loop does not cause continuous growth of the Java Virtual Machine stack. The operations involved in execution of the loop are simply assignments to registers, and non-recursive method calls. Consequently, we are now able to evaluate potentially infinitely-recursive expressions using an iterative process.

The process of evaluation presented in this chapter is essentially while loop. Hence, the execution of the interpreter no longer causes a growing Java control context, or continuation.

Chapter 7

Interpreter Optimisations

In Chapter 6 we developed an implementation of an interpreter that can evaluate any `JLambda` expression. The goal of that chapter was simply a working interpreter—no attention was paid to the performance of the interpreter. In this chapter we shift our focus to the interpreter's performance. We present four changes to the implementation of the interpreter; each change is an attempt to increase the interpreter's execution speed. The remainder of the chapter is an explanation of how we implement each of these potential optimisations.

7.1 Interned Strings

The first optimisation we perform is the interning of certain strings used throughout the implementation of the interpreter. As we explained in the Chapter 5, the form returned by the parser is a list in which each element is a Java `String` or a list. The evaluator determines the type of a form by examining the string at the head of the list—the form ID. The evaluator then dispatches on the form ID to a method responsible for evaluating forms of the specified type.

Our initial implementation of the interpreter examined form IDs by used the Java's `String.equals` method to determine string equality. The following code illustrates how this was implemented in the interpreter:

```
class State {
    ...
    String formId = exp.car();
```

```
    ...
    public void step() {
        ...
        if (formId.equals("apply")) {
            ...
        }
    }
    ...
}
```

For expressions that may be evaluated more than once, such as closure bodies, the current implementation of the evaluator performs a string comparison of its form IDs every time it is evaluated. Since the form ID of a given form never changes, its repeated comparison is unnecessarily expensive.

We would like to reduce this expense by taking advantage of the fact that form IDs are constant for each expression. One way in which this can be done is by using Java's `String.intern` method. The `intern` method guarantees that when it is invoked on a string, only one copy of that string will exist. If `intern` is called on a string that is equal to another already-`intern`ed string, a reference to the existing string is returned. This allows us to ensure that only one copy exists of the string that represents a form ID, which in turn allows form ID strings to be tested for equality using the object equality operator, `==`. Naturally, the object equality operator is faster than the `String.equals` method.

The introduction of interned strings to the interpreter implementation requires, in addition to the replacement of the `String.equals` method with the '`==`' operator, the interning of any string that the parser determines is in the head position of a form. We implement this change by defining a new class, `SymbolTable`, that contains strings that represent all of the form IDs. The strings are stored in a hash table so that the parser can determine whether a given string is a valid form ID.

```
class SymbolTable {
    public static final String AGET        = "aget";
    public static final String AND         = "and";
    ...
    public static final String SINVOKE     = "sinvoke";
    public static final String TIMES       = "*";
```

```
    private static final HashMap FORMIDS = new HashMap();

    public static void init() {
        FORMIDS.put(AGET,       null);
        FORMIDS.put(AND,        null);
        ...
        FORMIDS.put(SINVOKE,    null);
        FORMIDS.put(TIMES,      null);
    }

    public static boolean contains(String s) {
        return FORMIDS.containsKey(s);
    }
}
```

The following code reflects the relevant changes to the parser:

```
class List {
    public static List parseCdr(Parser parser, ...) {
        ...
        String tok = Parser.nextToken(parser);
        if(tok.equals("(")){
            ...
        } else if(tok.equals(")")){
            ...
        } else {
            Object exp = parseToken(tok);
            if (exp instanceof String) {
                String s = exp.toLowerCase();
                if (SymbolTable.contains(s)) {
                    exp = s.intern();
                }
            }
            ...
```

```
        }
    }
}
```

The changes to the evaluator necessary to take advantage of interned form ID strings are shown below:

```
class State {
    ...
    String formId = exp.car().toLowerCase();
    ...
    public void step() {
        switch(mode) {
        case EVAL:
            ...
            if (formId == SymbolTable.APPLY) {
                ...
            }
       }
       ...
    }
    ...
}
```

Since the comparison of the strings that represent form IDs is a core operation of the evaluator, we expect that interning such strings will increase the interpreter's execution speed.

7.2 Hash Table Dispatch

The interpreter obtained by applying the optimisation described in Section 7.1 , although using a more efficient test for string equality, still uses a series of `if-else` comparisons when dispatching on the form ID. We can perhaps achieve a further optimisation by replacing the sequence of `if-else` statements by a more efficient form of dispatch.

One way to replace the `if-else` dispatch procedure is to hash the form IDs, and then retrieve from a hash table the method responsible for handling the evaluation of forms of the type specified by the form ID. We implement this version of form ID dispatch by defining, for each form type, a class that contains a single method, `eval`, for handling the evaluation of forms of that type. The hash table is populated with a single instance of each class during initialisation of the interpreter. When a form is evaluated, the interpreter retrieves from the hash table the object corresponding to the that form, and invokes the object's `eval` method.

The following is the definition of the class responsible for handling the evaluation of `apply` forms:

```
class LambdaFormHandler {
    public Object eval(List body, Environment env) {
        return Evaluate.evaluateLambda(body, env);
    }
}
```

The interpreter creates an instance of each of these classes and stores them in the hash table defined in the `SymbolTable` class:

```
final class SymbolTable {
    public static final String AGET        = "aget";
    public static final String AND         = "and";
    ...
    public static final String SINVOKE     = "sinvoke";
    public static final String TIMES       = "*";

    private static final HashMap FORMIDS = new HashMap();

    public static void init() {
        FORMIDS.put(AGET,       new AgetFormHandler());
        FORMIDS.put(AND,        new AndFormHandler());
        ...
        FORMIDS.put(SINVOKE,    new SinvokeFormHandler());
        FORMIDS.put(TIMES,      new TimesFormHandler());
    }

    public static boolean contains(String s) {
```

```
        return FORMIDS.containsKey(s);
    }

    public static FormHandler get(String formId) {
        return (FormHandler) FORMIDS.get(formId);
    }
}
```

The introduction to the interpreter of hash table dispatch requires only straight-forward changes to the interpreter's evaluation loop: the sequence of `if-else` statements is replaced by a single hash table lookup to retrieve the object that knows how to evaluate the specified form, followed by invocation of the retrieved object's `eval` method.

```
class State {
    ...
    String formId = exp.car();
    ...
    public void step() {
        switch(mode) {
        case EVAL:
            ...
            SymbolTable.get(formId)).eval(this);
            break;
       }
       ...
    }
    ...
}
```

7.3 Continuation Pools

The transformation of the interpreter into continuation-passing style, as described in Chapter 6, necessitated the introduction of a `Continuation` class for each form type. During the evaluation of a given form, a `Continuation` object is constructed for the evaluation of every sub-form of that form. Upon completion of the evaluation of each sub-form, its `Continuation` object is promptly discarded. Therefore, a consequence of the way `Continuation` objects are used in expression evaluation is the creation and disposal of many `Continuation` objects during execution of the interpreter.

Since the majority of the `Continuation` objects are short-lived, it may be more efficient re-use them by maintaining object pools. The interpreter uses object pools in the following way: when a continuation object for a particular form type is required, it can be obtained from the pool of appropriate continuation objects; when the object is no longer required, it is returned to the pool from which it was obtained.

Adding continuation pools to the interpreter is straight-forward. For each continuation class we define a pool manager class, whose responsibility is to allocate and reclaim the corresponding continuation objects. At startup, the pool manager creates an initial pool of continuation objects; when it receives a request for an object it returns a reference to an unused continuation object in the pool. If all objects in the pool are used when the pool manager receives a request, the pool manager enlarges the pool, populating it with new continuation objects; it then returns a reference to one of the newly-created objects.

The changes to the evaluator necessary to enable it to use continuation pools are twofold. First, we must replace each statement in which a continuation object is created with one that requests a continuation object from the appropriate continuation pool manager. Second, we must locate each place in the program where a continuation object "exits" (and is discarded), and insert there code to return the object to the pool for reuse.

With the addition of continuation pools to the interpreter implementation, we hope to have reduced the cost of creating and disposing of continuation objects, by reusing them as much as possible.

7.4 Replacing Variables with Lexical Addresses

The current implementation of lexical environments is a simply a list of bindings. In such an implementation, lookup of lexical variables involves traversing the list of bindings, and comparing the variable in each binding to the target variable. Although this implementation has the virtue of simplicity, it is inefficient: the time taken for variable lookup is linear in the size of the program. In this section we describe an implementation of environments that does not rely on search, but instead relies on the replacement of variables in expressions with the variable's lexical address.

Because `JLambda` is lexically scoped, the environment in which an expression is evaluated will have a structure identical to the lexical structure of the program in which the expression appears. The following expression serves as an example:

```
(let ((a (int 0))
      (b (int 1)))
  ((lambda (w x y)
     (let ((y (int 2))
           (z (int 3)))
       (+ x y b)))
   (int 4) (int 5) (int 6)))
```

At the point when the interpreter evaluates the expression `(+ x y b)`, the environment has a structure represented by the following list of bindings:

$$\mathtt{y} \to 2, \mathtt{z} \to 3, \mathtt{w} \to 4, \mathtt{x} \to 5, \mathtt{y} \to 6, \mathtt{a} \to 0, \mathtt{b} \to 1$$

The structure of the environment during evaluation clearly mirrors the lexical structure of the example program. Thus, it is possible to predict the location in the evaluation environment of a particular variable simply by observing its location in the lexical structure of the program. In the example program, the variable `y` in the expression `(+ x y b)` is the first variable bound in the immediately enclosing lexical scope; its binding in the environment is located in first position. Similarly, `b` is bound by the second binding in an enclosing scope that is at distance 2 from the current scope; the binding for `b` in the environment is located in seventh position.

The correlation of the position of a variable in the environment with its position in the lexical structure of the program can be exploited to obtain a more efficient implementation of environment operations. Since the position of a variable in the environment can be determined statically, we can, prior to evaluation of an expression, replace each variable by what we know will be its address in the environment. Then, during evaluation of the expression, environment operations use these addresses to directly retrieve values from the environment.

Once we have made it possible to directly access the environment by address, we would like an implementation of environments that is optimised for such access. We therefore implement environments as a stack of frames. Each frame represents a lexical scope, and stores the values of the variables introduced by that scope. (Note that there is no longer any need to store the variables themselves.) The evaluator pushes a new frame onto the stack whenever a new lexical scope is created (by a `let` expression, for example). Since values will be accessed directly by environment address, the most efficient way to store them within a frame is in a vector. We call this environment implementation a `stack frame environment`.

Returning to the example program, when the interpreter evaluates the expression `(+ x y b)` the structure of the stack frame environment is represented by the following stack of frames:

$$[2\ 3] \rightarrow [4\ 5\ 6] \rightarrow [0\ 1]$$

In the example program, the variable `y` in the expression `(+ x y b)` is the first variable bound in the immediately enclosing lexical scope; accordingly, it's value, `2`, is stored in the first element of the first stack frame. Similarly, `b` is bound by the second binding in an enclosing scope that is distance 2 from the current scope; the value of `b`, `1`, is stored in the second element of the third stack frame.

7.4.1 Stack Frame Environments

With environments implemented as stack frame environments, it is natural that a lexical address should consist of two components: a frame number, which specifies how many frames to pass over, and a slot number, which specifies how many values to pass over in that frame. We implement lexical addresses with the class `LexicalAddress`.

```
class LexicalAddress {
    private int frame;
    private int slot;

    public LexicalAddress(int f, int s) {
        frame = f;
        slot = s;
    }

    public int getFrame() {
```

```
        return frame;
    }

    public int getSlot() {
        return slot;
    }
}
```

Environment frames are implemented by the `EnvFrame` class. An environment frame is a vector of values implemented as an array of `Object`. Primitive values are stored in the array via their wrapper objects. The `lookup` method simply returns the value stored in the vector at the position specified by the method argument.

```
class EnvFrame {
    private Object[] vector;

    public EnvFrame(Object[] v) {
        vector = v;
    }

    public Object lookup(int slot) {
        return vector[slot];
    }
}
```

Finally, we change the `Environment` class to implement a list of `EnvFrames`. The `lookup` operation takes a lexical address as an argument and returns the value at that address. `lookup` locates the correct frame by skipping the number of frames specified by the `frame` field of `address`, and uses the slot field of `address` to index into the frame.

```
class Environment extends List {
    public Environment() {
        super();
    }

    public Environment(Object[] vals, Environment cdr) {
```

```
        super(new EnvFrame(vals), cdr);
    }

    public Environment(EnvFrame frame, Environment cdr) {
        this.car = frame;
        this.cdr = cdr;
        this.empty = false;
    }

    public Environment extend(Object[] vals) {
        return new Environment(vals, this);
    }

    public Object lookup(LexicalAddress address) {
        int frame = address.getFrame();
        Environment ptr;
        for (ptr = this; frame > 0; frame--) {
            ptr = (Environment) ptr.cdr;
        }

        return ((EnvFrame) ptr.car).lookup(address.getSlot());
    }
}
```

With stack frame environments implemented, we must add to the interpreter functionality for replacing a variable in an expression with its lexical address. Since variable replacements must be performed before the evaluation of an expression begins, we introduce a new interpretation phase, the *syntactic analysis* phase, in which such operations are performed.

7.4.2 Syntactic Analysis

In the syntactic analysis phase of interpretation, we replace each occurrence of a variable in an expression with its lexical address. The syntactic analysis phase is implemented using the `analyse` method of the `Analyse` class. The `analyse` method takes as an argument the form produced by the parsing phase, and returns a new form identical to the original, except that each variable is replaced by its lexical address.

Analysis Environments

To determine the lexical address of each variable in a program, the `analyse` method must keep track of lexical scope as it analyses each expression in the program. It does this by using an environment, which we call an *analysis environment*, that represents the current lexical scope. An analysis environment is similar in structure to an evaluation environment; whereas an evaluation environment associates values to addresses, an analysis environment associates addresses to variables. When a new lexical scope is created (by a `let` expression, for example), a frame containing the newly introduced variables is pushed onto the stack of frames implementing the analysis environment. Thus, the lexical address of a variable in an expression is given by its address in the current analysis environment.

An analysis environment frame is implemented by the `AnalysisEnvFrame` class; the variables in the frame are stored in a vector. The `AnalysisEnvFrame` class provides a `contains` operation, for determining whether a frame contains a particular variable, and a `getSlot` operation, for calculating the position within a frame of a given variable.

```
class AnalysisEnvFrame {
    private String[] vector;

    public AnalysisEnvFrame(String[] v) {
        vector = v;
    }

    public boolean contains(String var) {
        for (int slot = 0; slot < vector.length; slot++) {
            if (var.equals(vector[slot])) {
                return true;
            }
        }

        return false;
    }
```

```
    public int getSlot(String var) {
        for (int slot = 0; slot < vector.length; slot++) {
            if (var.equals(vector[slot])) {
                return slot;
            }
        }
    }
}
```

The analysis environment is implemented by the `AnalysisEnv` class. The most important operation of the `AnalysisEnv` class is `lookup`, which, given a variable, returns the address in the environment of that variable. If the variable is not present in the environment, `null` is returned.

```
class AnalysisEnv extends List {
    public AnalysisEnv() {
        super();
    }

    public AnalysisEnv(String[] vars, AnalysisEnv cdr) {
        this(new AnalysisEnvFrame(vars), cdr);
    }

    public AnalysisEnv(AnalysisEnvFrame frame, AnalysisEnv cdr) {
        this.car = frame;
        this.cdr = cdr;
        this.empty = false;
    }

    public AnalysisEnv extend(String[] vars) {
        return new AnalysisEnv(vars, this);
    }

    public LexicalAddress lookup(String var) {
        int frame = 0;
```

```
        int slot = 0;

        AnalysisEnv cur = this;
        while (!cur.isEmpty()) {
            AnalysisEnvFrame f = (AnalysisEnvFrame) cur.car;
            if (f.contains(var)) {
                return new LexicalAddress(frame, f.getSlot(var));
            }
            cur = (AnalysisEnv) cur.cdr;
            frame++;
        }

        return null;
    }
}
```

The `analyse` Method

The `analyse` method receives a form from the parser, and constructs and returns a new form, which is identical to the original, except that each variable has been replaced by its lexical address. `analyse` is therefore implemented as a case analysis on the syntactic structure of the form to be analysed. For each case, `analyse` calls itself recursively on each subexpression; the new form is then assembled from the analysed subexpressions.

In the case where the expression is a string, `analyse` assumes the string denotes a lexical variable and attempts to retrieve the variable's address in the analysis environment. If the variable is not found in the environment, the string must denote either a global variable, a Java field, or an unbound variable. Since all of these possibilities are handled by the evaluator, `analyse` simply returns the string unchanged.

```
class Analyse {
    public static Object analyse(Object exp, AnalysisEnv env) {

        if (exp instanceof String) {
            LexicalAddress a = env.lookup((String) exp);
            return (a != null ? a : exp);
```

```
        }

        ...
    }
}
```

The analysis of + expressions provides an example of how slightly more complex expressions are analysed. First, the two argument subexpressions are analysed; then the resulting forms, along with the form ID, are combined and returned as the analysed + form.

```
class Analyse {
    public static Object analyse(Object exp, AnalysisEnv env) {

        ...

        List form     = (List) exp;
        Object formId = (String) form.car();
        List body     = form.cdr();

        if (formId.equals("+") {
            return new List(formId,
                            analyse(body.car(), env),
                            analyse(body.cadr(), env)),
        }

        ...
    }
}
```

The most interesting expressions are those that introduce new lexical scopes: the `let` and `lambda` expressions. The `analyse` method processes these expressions by first extending the current analysis environment with a frame containing the variables being introduced, and then analyses the body of the expression in the newly extended environment.

The following part of `analyse` shows how `let` expressions are analysed. Analysis begins with the construction of a new list of bindings, in which each expression in the value position is replaced with its analysed version. Following the analysis of each binding expression, the analysis environment is extended with a frame, via a call to `AnalysisEnv.extend`, consisting of the variable bound to that expression. We create a new frame for each variable so that it is visible in subsequent binding expressions. Next, the body of the let expression is analysed in the extended analysis environment. Finally, a new `let` form containing the analysed subexpressions is constructed and returned.

```
class Analyse {
    public static Object analyse(Object exp, AnalysisEnv env) {
        ...
        List form     = (List) exp;
        Object formId = (String) form.car();
        List body     = form.cdr();

        if (formId.equals("let") {
            List bindings = (List) body.car();
            List newBindings = new List();

            for (int n = 0; n < bindings.size(); n++) {
                List   b   = (List) bindings.nth(n);
                String var = (String) b.car();

                List newB = new List(var, analyse(b.cadr, env));
                newBindings = newBindings.append(new List(newB))

                env = env.extend(new String[] {var});
            }

            Object newBody = analyse(body.cadr(), env);
            return new List(formId, newBindings, newBody);

        }

        ...
   }
}
```

With these changes made, we have completed the implementation of the interpreter's syntax analysis phase. To give an example of the output produced by the analysis phase, we again return to the example program:

```
(let ((a (int 0))
      (b (int 1)))
  (apply (lambda (w x y)
     (let ((y (+ a b))
           (z (+ y (int 3))))
       (+ x y b)))
   (int 4) (int 5) (int 6)))
```

When this expression is passed to `analyse`, a form is produced in which all variables have been replaced by their lexical addresses. In the following notation, a lexical address is represented by the pair `{f, s}`, where `f` denotes the number of frames to skip, and `s` denotes the frame's slot number.

```
(let ((a (int 0))
      (b (int 1)))
  ((lambda (w x y)
     (let ((y (+ {2,0} {1,0})
           (z (+ {0,0} (int 3))))
       (+ {2,1} {1,0} {3,0})))
   (int 4) (int 5) (int 6)))
```

Adding Lexical Addresses to the Evaluator

If the evaluator is to evaluate forms produced by the syntax analysis phase, it must be altered to recognise and evaluate lexical addresses. Such an alteration requires only a simple addition to the `step` method of the `State` class: if an expression is a lexical address, we simply return the value at that address in the current environment.

```
public void step() {
    switch (mode) {
    case EVAL:
        if (exp instanceof LexicalAddress) {
```

```
            val  = env.lookup((LexicalAddress) exp);
            mode = RETURN;
            break;
        }
    ...
}
```

The only other modifications to the interpreter necessary for it to use lexical addresses are to the `LetCont` and `ApplyCont` classes, so they pass only an array of values to the `Environment.extend` method.

With these modifications, we have an interpreter that uses a separate syntax analysis and evaluation phase. In the syntax analysis phase, each occurrence of a variable in an expression is replaced with its lexical address in the program structure. These changes remove the need for environment operations to search the environment; environment operations now directly access the environment by address. In addition, we changed the implementation of environments from a simple linked list of bindings to a stack frame implementation. We expect these changes to yield a more efficient interpreter.

Chapter 8

Evaluation of Interpreter Optimisations

In this chapter we evaluate each of the potential optimisations of the interpreter implementation that were described in Chapter 7. We evaluate the changes to the interpreter implementation by using benchmark programs: we measure and compare the time taken by each version of the interpreter to evaluate a benchmark program.

We use two benchmark programs in our comparisons: `tak` and `clicker`. The `tak` benchmark is a simple program designed to test the core functions of the interpreter. The `clicker` benchmark, which was discussed extensively in Chapter 4, is a program designed to be more representative than `tak` of realistic `JLambda` programs.

All of the measurements are performed on an unloaded machine with a 2.4GHz Intel Celeron CPU and 256MB of main memory. All Java programs are compiled and run using version 1.4.2_04 of Sun's Java Development Kit. To measure the execution time of an interpreter, we use the GNU `time` command to measure the total CPU time used by the interpreter while evaluating a benchmark program. We record the execution time of each interpreter-benchmark pair by performing the measurement 10 times and taking the mean of the measurements.

We perform benchmark measurements for five versions of the interpreter, each with different optimisations:

- No optimisations
- Interned strings

- Interned strings and hash table dispatch
- Interned strings, hash table dispatch and continuation pools
- Interned strings and syntax analysis

8.1 The `tak` Benchmark

`tak` is Richard Gabriel's publicly-available[7] implementation of the Takeuchi function. The following is an implementation of `tak` in `JLambda`:

```
(define tak (x y z)
  (if (not (< y x))
      z
      (apply tak (apply tak (- x (int 1)) y z)
                 (apply tak (- y (int 1)) z x)
                 (apply tak (- z (int 1)) x y))))
```

Examination of the `tak` function shows that it measures primarily (recursive) function calls and environment operations. Because `tak` is function-call heavy, it provides a good test of function call, recursion and environment operations. However, because it does little besides function calls—it does not make use of the Java runtime facilities, for example—it is not representative of most `JLambda` programs.

We include in our tests an implementation of `tak` in Java, and compare its execution time to those of of the `JLambda` interpreters. The following Java class implements the `tak` function:

```
class Tak {
    static int tak(int x, int y, int z) {
        if (x <= y) {
            return z;
        } else {
            return tak(tak(x - 1, y, z),
                       tak(y - 1, z, x),
                       tak(z - 1, x, y));
        }
```

```
    }
}
```

8.1.1 Performance Results

We test the five versions of the interpreter with six versions of the `tak` benchmark program. Each benchmark program differs in the arguments that are passed to the `tak` function. The Java implementation of `tak` is tested with the same sets of arguments that are used in the other benchmark programs.

The six versions of the benchmark program are summarised in Table 8.1.

Benchmark Number	Invocation of the `tak` Function
1	`(apply tak (int 9) (int 6) (int 3))`
2	`(apply tak (int 12) (int 8) (int 4))`
3	`(apply tak (int 15) (int 10) (int 5))`
4	`(apply tak (int 18) (int 12) (int 6))`
5	`(apply tak (int 21) (int 14) (int 7))`
6	`(apply tak (int 24) (int 16) (int 8))`

Table 8.1: Summary of the `tak` benchmarks

Figure 8.1 is a graph of the execution times of the `tak` benchmark programs for each interpreter version, along with the execution times of the Java versions of the `tak` benchmark.

The performance results show that by interning the strings that store form IDs, we have increased the speed of the interpreter. Similarly, the addition of the syntax analysis phase, along with the replacement of variables with lexical addresses, has significantly increased execution speed. However, both the addition of hash table dispatch and the addition of continuation pools caused a decrease in the execution speed of the interpreter. We discuss these results further in Section 8.3.

The conclusion we draw from `tak` benchmark performance results is that the most efficient interpreter implementation of those tested is one that uses interned strings for form IDs, and which uses a separate syntax analysis phase in which variables in expressions are replaced by their lexical addresses.

The performance results also demonstrate that, not surprisingly, programs implemented in `JLambda` execute significantly slower than equivalent programs implemented in Java.

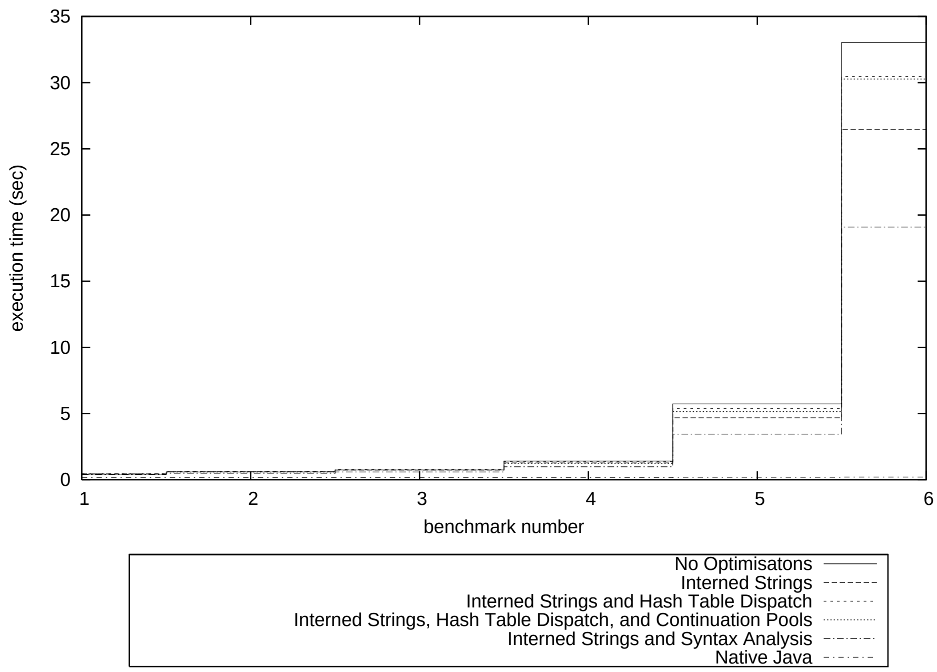

Figure 8.1: Execution times of the `tak` benchmark

8.2 The `clicker` Benchmark

For this benchmark we use `clicker`, which was presented in Chapter 4. `clicker` is a small `JLambda` GUI program that uses Java's Swing library to display a window; when the mouse is clicked in the window, a small coloured circle is placed at the point of the click. The implementation of `clicker` makes heavy use of Java's run-time facilities; for example, `clicker` registers event handlers by creating closures and assigning them to mouse click events.

Since `clicker` requires mouse input we have written a driver program in Java to provide the necessary input. The driver program uses the `java.awt.Robot` class to send a specific number of mouse click events to `clicker`'s window. Additionally, the rate at which the mouse click events are submitted to the `clicker` program is held constant across all benchmarks.

We include here the source code for the driver program. The following version of the program delivers 84 mouse clicks to the `clicker` program.

```
import java.awt.AWTException;
import java.awt.Robot;
```

```
import java.awt.event.InputEvent;
import java.io.IOException;

public class ClickerBenchmark {
    public static void main(String[] args)
        throws AWTException {
        String version  = args[0];

        String command = "java -ea -cp " + version + " "
            + args[1] + " " + version + "Scheme/clicker.lsp";

        try {
            Runtime.getRuntime().exec(command);
        } catch (IOException e) {
            e.printStackTrace();
        }

        Robot robot = new Robot();

        /*
         * Before generating an event, wait until all events
         * currently on the event queue have been processed.
         */
        robot.setAutoWaitForIdle(true);

        // Wait long enough for frame to appear.
        robot.delay(3000);

        int x = 45;
        int y = 100;
        for (int d = 0; d < 420; d += 5) {
            robot.mouseMove(x + d, y + d );
            robot.mousePress(InputEvent.BUTTON1_MASK);
            robot.mouseRelease(InputEvent.BUTTON1_MASK);
        }
```

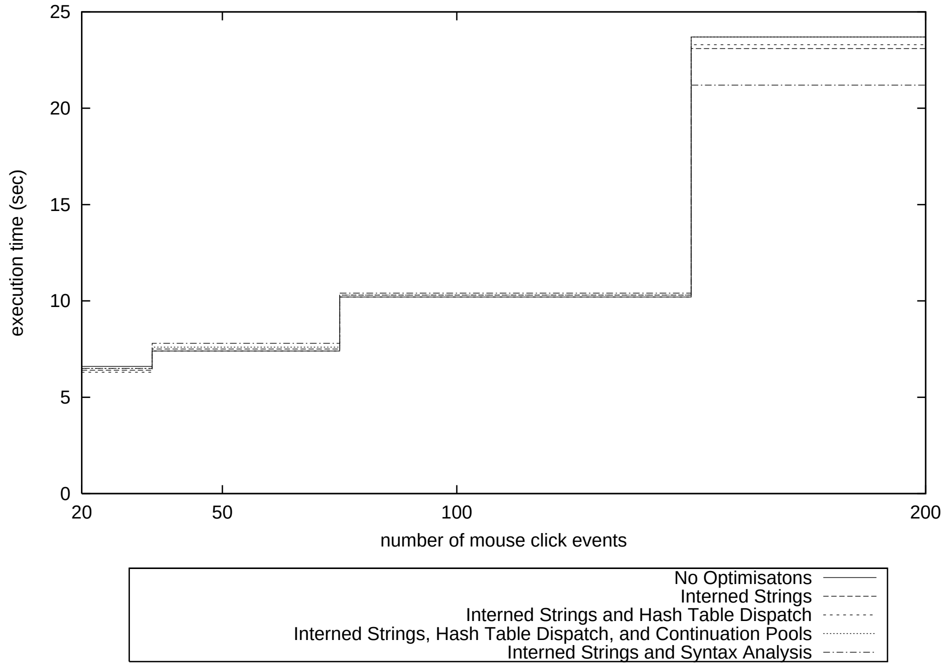

Figure 8.2: Execution times of the `clicker` benchmark

```
    }
}
```

8.2.1 Performance Results

We test the five versions of the interpreter with four different invocations of the `clicker` benchmark program. Each invocation differs in the number of mouse click events that are sent to `clicker` by the driver program. We measure the interpreter performance with 20, 50, 100 and 200 click events.

Figure 8.2 is a graph of the execution times of the `clicker` benchmark programs for each interpreter version.

The results indicate that there is very little difference in the performances of the interpreters when executing the `clicker` benchmark program. The only exception is the interpreter that uses a syntax analysis phase—it shows superior performance as the number of mouse click events increases. Since most of the operation of the `clicker` program is handled by the Java Swing library, the role of the interpreter in execution of the program is comparatively small. Once the Swing objects have been created and displayed, the execution of the program is controlled by the Swing library's event loop. The only time `JLambda` expressions are evaluated by the interpreter during the event loop is when a closure is invoked in response to a mouse click event. Since the closure that creates a node in the `clicker` program is a relatively simple expression, we expect the performance of the various interpreter versions to be similar.

The `clicker` benchmark performance results, like the `tak` benchmark performance results, demonstrate that the most efficient interpreter implementation of those benchmarked is one that uses interned strings for form IDs, and which uses a separate syntax analysis phase, in which variables in expressions are replaced by their lexical addresses.

8.3 Discussion of Performance Results

In this section we briefly discuss the benchmark performance results for each attempted optimisation of the interpreter implementation.

8.3.1 Interned Strings

It is not surprising that the introduction of string interning to the interpreter has increased its execution speed. Interning strings allows us to compare two strings by using object equality operator, `==`, instead of the `String.equals` method. The `==` operator tests for object equality by checking whether two object references point to the same object; the `String.equals` methods tests whether two strings are equal by comparing them character by character. Clearly, the object equality operation is faster than a call to the `String.equals` method.

8.3.2 Hash Table Dispatch

The addition of hash table dispatch, however, has decreased the execution efficiency of the interpreter. We conclude that replacing string comparison dispatch by hash table dispatch in the interpreter has introduced more operations than it has saved. Hash table dispatch involves several operations: the hashing of a form ID; the retrieval of an object from a hash table; and the invocation of the object's form-handler method. In contrast, string comparison dispatch involves only a number of if-else comparisons using object the equality operator, and the invocation of a form-handler method.

8.3.3 Continuation Pools

The situation for continuation pools is analogous to that of hash table dispatch. The addition of continuation pools to the interpreter has introduced more operations than it has saved. The performance results suggest that the book-keeping necessary to manage pools of continuations is more expensive than the costs of on-demand creation and reclamation of objects.

8.3.4 Addition of Syntax Analysis

The replacement of variables with lexical addresses has yielded an increase in the interpreter's performance. Such a performance increase is consistent with our expectations. Although the addition of the syntax analysis phase introduced some execution overhead to interpreter, it allowed us to remove many of the operations involved in variable lookup during the evaluation phase. Since the operations in the syntax analysis phase are performed only once, while the operations in the evaluation phase may take place many times, we have achieved a net gain in the execution efficiency of the interpreter.

Chapter 9

Conclusions and Suggestions for Further Work

We have presented in this thesis an implementation of an interpreter for the `JLambda` language. In addition, we have suggested, implemented and evaluated four potential optimisations of the implementation of the interpreter. Our performance measurements showed that the greatest improvements to interpreter's performance were achieved by interning the strings representing the form IDs, and by introducing a syntax analysis phase, in which variables in expressions are replaced by their lexical addresses.

The most important conclusion is that it is possible to implement, in a reasonably straight-forward manner, an interpreter for a recursive target language using Java (a non-tail recursive language) as the implementation language. Java's object-oriented features enable the continuation-passing transformation and register-machine transformation, both of which are crucial to implement a recursive interpreter in a non-tail recursive language, to be implemented conveniently.

A second conclusion is that the addition to the interpreter of a syntax analysis phase, in which variables in expressions are replaced by their lexical address, yields a significant increase in the interpreter's performance. The addition of a syntax analysis phase is a process that is not difficult, and does not complicate the interpreter's implementation. Rather, since it separates the process of evaluation into two distinct phases, the addition of a syntax analysis phase yields an interpreter implementation that is simpler, and consequently easier to understand and modify.

There are many more improvements to the interpreter implementation that are possible.

For example, much of the error checking can be moved from the evaluation phase to the syntax analysis phase.

A further improvement to the interpreter implementation would be achieved by choosing a more efficient representation of expressions during evaluation. One possibility is the representation of `JLambda` expressions by Java objects. The advantage of this representation is that dispatch on expression type could use the `instanceof` operator, which, we expect, would afford a faster dispatch operation than the methods presented in this thesis, so achieving the hoped-for benefits of implementing expressions as objects.

Bibliography

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

[2] K. Anderson, T. Hickey, and P. Norvig. SILK: A Playful Blend of Scheme and Java. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 13–22, September 2000.

[3] Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proceedings of the 1988 ACM Symposium of LISP and Functional Programming*, pages 86–95. ACM Press, 1988.

[4] Per Bothner. Kawa – Compiling Dynamic Languages to the Java VM. In *Proceedings of the Usenix Annual Technical Conference*, June 1998.

[5] William D. Clinger. Proper Tail Recursion and Space Efficiency. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 174–185. ACM Press, 1998.

[6] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. PhD thesis, Indiana University, August 1987.

[7] Richard P. Gabriel. *Performance and Evaluation of LISP Systems*. MIT Press, 1985.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[9] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[10] I. A. Mason and C. L. Talcott. IOP: The InterOperability Platform & IMaude: An Interactive Extension of Maude. In *International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2004.

[11] Scott G. Miller. SISC: A Complete Scheme Interpreter in Java. Technical report, Indiana University, January 2002.

[12] G. Plotkin. Call by Name, Call by Value, and the Lambda Calculus. *Theoretical Computer Science*, 1, 1974.