Design and Implementation of Exception Handling with Zero Overhead in Functional Languages

By

Ramon Zatarain Cabada

A dissertation
submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment of the requirements for
the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
May, 2003

Abstract

Title: Design and Implementation of Exception Handling with Zero Overhead in Functional Languages
Author: Ramon Zatarain Cabada
Major Advisor: Ryan Stansifer, Ph.D.

This dissertation considers the implementation of exception handling specifically for functional languages. Some implementations incur overhead for using exception handling even when no exceptions are raised. We show the results of some experiments with the SML of New Jersey and OCAML compilers, two well-known compilers for functional languages. Imperative languages avoid this overhead by using tables, but the approach does not easily transfer to compilers using continuation passing style (CPS). This dissertation proposes an approach that works with CPS compilers like SML of New Jersey.

We first present an experiment where programs in SML are written with and without exception handlers. From these results, we conclude that programs with exception handling produce overhead even when no exceptions are raised. Then, we analyze the source of the exception handling overhead in the SML of New Jersey compiler. We present a solution to the problem. The new approach uses two continuations instead of the one continuation. One continuation encapsulates the rest of the normal computation as usual. A second continuation is used for passing the abnormal computation. The second continuation is not passed as an extra argument but is passed as a displacement from the first continuation.

We have implemented a basic CPS compiler for functional languages with exception handling. With it we were able to implement the new approach to exception handling and compare it side-by-side with the approach taken by the SML of New Jersey compiler. We show that the new approach to exception handling adds no overhead to the normal flow of control.

The importance of our new approach to exception handling for CPS compilers proposed in this dissertation is the improved run-time performance in every case in which an exception handler is used.

Table of Content

List of Figures

Acknowledgment

First and foremost I would like to thank my advisor Dr. Ryan Stansifer for all the effort he spent in guiding my dissertation. He not only inspired me to work in functional languages but also help to improve my abilities as a programmer, as a writer, and most at all as a computer scientist. I also highly appreciate his company and friendship during this last five years.

I would also like to acknowledge my committee, Dr. Phil Bernhard, Dr. Pat Bond, and Dr. Dennis Jackson for theirs valuable comments about this research. Special thanks to Dr. George (Jorge) Abdo for his words and advice he provided during this last five years.

I also thank many faculty members from the Florida Institute of Technology for providing insight and comprehensive explanations to their many fields of expertise.

Finally, I want to give my gratitude to my wife for her support not only as a wife but also as a computer professional. She not only gave me her love and understanding but also gave me valuable comments and suggestions during my research.

Dedication

This dissertation is dedicated to the memory of my father

Rosalio Zatarin Osuna (1930-2003)

to my mother Delia T. Cabada Amarillas

to my wife Lucia and

to my three precious daughters Zyanya, Ana, and Naomi

1 Introduction

The value of exception handling is well-known in the field of software engineering. The first high-level language to have a mechanism for exception handling was PL/I [Ans76]. Before that, a common form of processing exceptions or error conditions was made by using IF statements inside the normal code in order to check the return code of some operations. When an exception occurred, normal processing activities were ended.

PL/I allowed a programmer to handle and propagate exceptions dynamically. The exceptions were associated to statements (today most programming languages like Ada and Java associate it to blocks of code). When an exception was raised the flow of control continued at the beginning of the statement which raised the exception.

A very important paper on exception handling was written for Goodenough [Goo75]. This paper describes a notation for an exception handling mechanism. Today, many models of exception handling in different programming languages are based in that notation.

Another language which pioneered exception handling facilities was CLU [LS79]. This programming language associated exceptions with blocks of code (procedures). A drawback in CLU was that exception propagation did not existed, and that exception was handled by the calling procedure.

After PL/I and CLU, a substantial amount of work has been done in programming languages to design alternative methods of exception handling. Ada [BR86], COMMON LISP [Ste84], SML [MTH90], Modula-3 [CDGJKN], C++ [Str91], and Java [GJS96] all support exception handling techniques.

There are several advantages to supporting exception handling in a language. One is to avoid cluttering programs with code for detecting error conditions. Another is to allow an exception to be propagated in its dynamic chain of calls. That provides a caller the possibility of knowing about the failure of an operation and is named dynamic propagation. But, most important is that the language encourages programmers to consider all events that can throw an exception during program execution.

Exception handing is very often the most important part of the system because it deals with abnormal situations. For a variety of reasons, not least among which is the fact that more than half of the code is often devoted to exception detection and handling, many failures are caused by the incomplete or incorrect handling of these abnormal situations. The requirements for correct system behavior during exception handling are in some sense even greater than for the system operating in normal mode.

1.1 Overview of the problem

When declaring and using exception handling the syntax and semantic of a language is pretty much the same. On the other hand, when we talk about efficiency we come to different results. Compilers of imperative languages like Java, Ada, and C++ implement exception handling without imposing overhead on normal execution [LYKPMEA, BR86, and Din00]. When a program defines an exception handler, the runtime performance of that program would be the same without exception handler definition. We can say that there is no runtime penalty for defining an exception handler which is never used. In other words, no runtime overhead occurs in the case in which no exceptions are raised. However, compilers of functional languages like SML/NJ [AM91] or CAML [Ler00] produce code that has exception handling overhead. We made some experiments in order to verify this. We found also the source of the overhead in the SML/NJ [AM91] compiler.

In this dissertation we present a new approach to implementing exception handling in functional programming languages. The new approach incorporates a method for implementing exception handling without imposing overhead on normal execution. In order to test the new approach, we had first to build a compiler for a functional language, where we implemented the two approaches: the traditional approach (the one used in the SML/NJ compiler), and the new approach proposed in this dissertation.

1.2 Outline of dissertation

Some material about functional languages and programming in them, especially SML, would be helpful for a reader of this dissertation [Har98, Hen80, Pau91, and Ull98].

Chapter 2 covers introductory and support material in functional programming, continuation-passing style, and exception handling. The material is presented with explanations and some code in different languages.

Chapter 3 describes the design and implementation of a model of translation and execution of programs. The translator generates CPS (continuation-passing style) programs, which are executed by an evaluator of CPS code. The chapter presents a set of examples used in testing the compiler in SML and CPS code.

Chapter 4 presents the abstract machine used to interpret the target code produced by the translator. We also show some examples of programs produced and tested in the machine.

Chapter 5 explains how the SML/NJ compiler implements exception handling. It also shows the experiments that we made in order to verify that the SML/NJ and OCAML compilers produce overhead in programs with exception handlers. Last, we explain the source of the overhead in SML programs by using an example.

Chapter 6 presents the new approach for exception handling implemented in our compiler. We start by describing a method used in imperative languages. This method uses a table of assembly code regions. Then, we explain the first part of the new approach where code produced by the compiler contains two continuations, and last we explain how zero overhead can be reached by doing some modifications to that approach.

Chapter 7 shows the experiments we did in order to test the performance of the new approach. First, it explains the experimental methodology and examples used in the tests. Second, it presents the results of performance of the programs using the old and new approach. Last, it makes an analysis of the results.

Finally, chapter 8 describes some conclusions of the work and the future research to be done.

2 Related Work

This chapter sets the stage for the presentations in chapter 3, 4, and 5. First, we review the fundamental concepts of functional programming; then we present an introduction of continuation passing style (CPS); finally, we describe exceptions in modern programming languages like SML, Java, and Ada.

2.1 Functional Programming

Functional languages

Functional languages focus on data values described by expressions (built from function applications and definitions of functions) with automatic evaluation of expressions. Programs can be viewed as descriptions declaring information about values rather than instructions for the computation of values or of effects [Rea89]. In functional programming languages there is no distinction between statements and expressions; names are only used to identify expressions and functions (and not memory locations); like in imperative languages they allow functions to be passed as arguments to other functions, or returned as results (*higher-order* functions) [WM95].

Functional Languages are divided in eager and lazy functional languages. In eager functional languages, the evaluation of arguments in a function application is made before the function is applied. This gives as a result that the same expression can be evaluated more than one time. On the other hand, lazy functional languages evaluate expressions in a demand drive way named **call by need**. In this technique the arguments of functions are evaluated once at most [Rea89]. For example, assuming **double** is defined by:

fun double x = plus x x

The evaluation of double (fact 5) begins with:

Double (fact 5) = plus (fact 5) (fact 5)

With call by need, **fact 5** needs to be evaluated only one time.

Examples of eager functional languages are Scheme [SSJ78], SML [MTH90], and CAML [Ler00]. Miranda [Tur85], Lazy ML [Aug84], Ponder [Fai82], and Haskell [Hud90] are all examples of lazy functional languages.

Eager functional languages

Functions that are always undefined with undefined arguments are called strict. A functional language that uses strict evaluation (to evaluate the arguments of the functions) is named eager (or strict) language. This evaluation technique in conventional programming languages like Pascal and C have been called **call by value**, while for functional languages is also called **applicative reduction order**. Some of the functional programming languages that use eager evaluation are ML, Scheme, and CAML.

The ML and SML programming languages

ML [MTH90] is a general-purpose programming language designed for large projects. It was developed in the late 1970s as the Meta-Language of the Edinburgh Logic for Computable Functions (LCF) theorem-proving system. It is an eager and functional programming language where current statements as blocks, conditional, assignments, etc. are encapsulated as expressions. Every expression has a statically determined type and will only evaluate to values of that type. Standard ML of New jersey (abbreviated **SML/NJ**) is a compiler and programming environment for ML written in ML with associated libraries, tools, and documentation [AM91, SML03]. SML/NJ is free, open source software. The core of the SML/NJ system is an aggressively optimizing compiler that produces native machine code for most commonly used architectures: x86 (IA32), Sparc, MIPS, IBM Power 1 (PowerPC), HPPA, and Alpha [SML03]. The compiler translates a source program into a target machine language program in several phases.

Lazy functional languages

Functions that can give defined results even when arguments are undefined are called non-strict. For example the expression (3 = 3) **or** ((5 div 0) = 4) might give a defined result of **true** (true **or** x = true), if we ignore the result of the second sub-expression (undefined). A functional language that uses non-strict evaluation (to evaluate the arguments of the functions) is named lazy (or non-strict) language. A program using lazy evaluation will not evaluate any expression unless its value is demanded by some other part of the computation. Lazy evaluation, also called call-by-need, is a modification of call-by-name that never evaluates the same **thunk** (sub-expression) twice. In this technique, lambda expressions are represented by graphs.

Lambda Calculus

Lambda calculus is a mathematical calculus for computable functions proposed by Alonzo Church in 1941 in order to establish the limits of what was computable. It is the theoretical foundation of functional languages. Next, we have a definition of a context free grammar for lambda calculus [Sta95]:

V	-->	x | y | z | v |…..	{Variables)
F	-->	a | b | c | f | g |….	{Set of functional symbols}
L	-->	V | F | λ V.L | L L | (L)	

Examples: x, f x, f (g x), λx.b, f (λ x.b), λy.(g x)

Lambda calculus provides a behavioral explanation of terms in the form of rewriting or reduction rules [Rea89]. They are used to simplify lambda terms. The most important reduction is the (β) reduction rule. This is given by:

Beta reduction (β): (λv.E1)E2 ====> E1[E2/v]

It means that the bound variable **v** in the body **E1** of the abstraction is substituted by the argument **E2**. It can be interpreted as saying that a function

λv.E1 applied to the actual argument **E2** means the same as the body **E1** where all occurrences of the formal parameter **v** have been replaced by **E2**. The variable **v** is bound to **E2**. This is essentially the call-by-name rule in ALGOL 60.

Examples:			
	(λ x. + x x) 4	====>	+ 4 4
	(λ v.v) c	====>	c
	(λ x. λ y. * y x) 3) 7	====>	(λ y. * y 3) 7
	(λ y. * y 3) 7	====>	* 7 3

A very important note is that we can represent any element of a programming language like a number, a primitive operation, a selection, recursion, etc. in lambda calculus.

Combinators

A free variable (non-local variable) affects the efficiency of a program (binding of free variables with its values). Combinators are a technique for transforming lambda expression into expressions that only include applications and combinators. These expressions are called closed expressions (expressions with no free variables). The idea is first, to translate a lambda expression into a combinator term in which no variables appear, and second, to translate a combinatory term into a combinatory using an algorithm named bracket abstraction.

Example: Consider the lambda expression (λ x. + x 1). It will be transformed to a combinatory term and then into a combinator as follow [Sta95].

```
(λ x. + x 1)                ==========>
[x] (+ x 1)                 ==========>{combinator term)
S ([x] (+ x)) ([x] 1)       ==========>{use of bracket abstraction}
S (S ([x] +) ([x] x)) [x] 1 ==========>{use of bracket abstraction}
S (S (K +) I) (K 1)                    {combinator}
```

Where the rules of evaluation are:

```
S x y z   ====> x z (y z)
K x z     ====> x
I x       ====> x
```

Example: We evaluate the corresponding lambda expression and combinatory and we get the same result.

Lambda expression	**Combinator**
(λ x. + x 1)	S (S (K +) I) (K 1)
(λ x. + x 1) 3	S (S (K +) I) (K 1) 3
(+ 3 1)	S (K +) I 3 (K 1 3)
4	(K + 3) (I 3) (1)
	+ 3 1
	4

Something remarkable to establish is that the third combinatory **I** can be built with combinators **K** and **S**. Then, any program can be described entirely by these two primitives.

Closures

In languages such as C without nested procedures, the run-time representation of a function value can be the address of the machine code for that function [App98]. This address can be passed as an argument, stored in a variable, and so on; when it is time to call the function, the address is loaded into a machine register, and the "call to address contained in register" instruction is used.

But this will not work for nested functions, where a function can return a function as a value (high-order functions). The problem is the non-local environment can not be found without the static link. The solution is to represent a function-variable as closure: a record that contains the machine-code pointer and a way to access the necessary free (non-local) variables. One simple kind of closure is just a pair of code pointer and static link; following the static link can permit access to the non-local variables. The portion of the closure giving access to values of variables is often called the **environment**.

Tail recursion

A function call **f(x)** within the body of another function **g(y)** is in tail position if "calling **f** is the last thing that **g** will do before returning". Applications of functions in tail positions can be implemented by direct jumps instead of the more sophisticated context switches needed for other function calls in most language implementations.

Example: (non-tail recursive function):

```
fun fact (n)=
        if  n=0 then 1 else fact(n-1) * n
```

The call to function **fact** is not the last operation of the function (the last operation is the multiplication operation), and then this is a non-tail recursive function.

Example: (tail recursive function):

```
fun fact (n)=
        if n=0 then 1 else n * fact(n-1)
```

The call to function **fact** is the last operation in the function.

2.2 Continuation-passing style (CPS)

Definitions of CPS

The purpose of CPS is to make every aspect of control flow and data flow explicit [App92].

It combines tail recursion with extra parameters (the continuations) in a function. In section 2.1.5 we showed an example that shows a transformation of a non-tail recursive function into a tail recursive function by using a continuation named **k**. Into the continuation we pass the context of the function (instead of using a stack for saving). Then the function does not need to return for doing last operations.

Another way of looking at CPS is as "a style of programming in which every user function **f** takes an extra argument **k** known as a continuation. Whenever **f** would normally return a result **r** to its caller, it instead returns the result of applying the continuation to **r**. The continuation thus represents the whole of the rest of the computation" [FOL].

Example: The program,

if x < 0 then x else f(x)

can be broken up into the expression

x < 0

and the continuation (expressed here as an evaluation context)

if [ ] then x else f(x)

The expression is evaluated and then passed into the continuation, which takes it the rest of the way.

Writing the continuation as a function, we can transform this program into:

(λv. if v then x else f(x))(x < 0)

Applying this transformation to every part of the program, we produce a program in continuation-passing style (CPS).

Thus, CPS is a style of writing programs where the events in the future, i.e. the rest of the computation, is passed as an explicit parameter. The value passed as this parameter is the **continuation**. A continuation is a procedure that takes the value of the current expression and computes the rest of the computation. Procedures do not return values; instead, they invoke the continuation with the result. If a program is fully CPS converted then there are no procedure return. Every procedure call is tail call, and the program's **control memory** is not stored in some invisible stack but explicity as the continuation.

Advantages of CPS

- No need for a stack (values go on and off the stack too many times), because functions never return.
- Every intermediate value of a computation is given a name (possible corresponding to a machine register). This allows an easy translation to machine code.
- Functional language compilers use CPS to transform the structure of the function from a lambda calculus form to an imperative form. We can then, apply conventional techniques like code optimization and generation to the transformed form.
- Beta reductions and others optimizations are easier to do too.

How can CPS be useful?

CPS can be useful in several different ways. When there are two or more possible continuations (one for success and one for failure), it is more convenient to let the procedure choose between its continuations than to force its caller always to perform a test based on some returned result, particularly since the nature of the information that must be returned depends on how the computation is to continue. Another use of **CPS** is to return multiple results by passing them to a multiple-argument continuation. This is often better than returning some data structure out of which the caller of the procedure would have to extract the values.

2.3 Exceptions

Exception handling techniques

An exception is the union of **error**, **exceptional case**, **rare situation**, and **unusual event** [LS98]. The entity that is raising an exception stops and waits for the completion of the exception processing. Exceptions are usually divided into two classes: predefined and user-defined. Predefined exceptions are declared implicitly and are associated with conditions that are detected by the underlying hardware or operating system; they are also called system-defined exceptions. In any case languages with exception handling allow the program to regain control.

The idea of exception handling is seen as the immediate response and consequent action taken to handle the exceptions. An exception handler is the code attached to (or associated with) an entity for one or several exceptions and is executed when any of these exceptions occur within the entity. Depending on the exception-handling mechanism, an entity can be a program, a procedure, a statement, an expression, an object, or data. Exception handling can be embedded into the operating system or into a programming language.

Exception handling in programming languages

Goodenough's notation is the first structured exception-handling mechanism proposed [Goo75]. It either terminates or resumes the program's execution after an exception is handled. If an exception is raised from an operation, the mechanism first tries to find local handlers, which is in the same context as the operation.

The programming language CLU [Lis79] is based on a simple model of exception handling and can support termination. The mechanism searches only one level up besides the local context. If a user wants to raise an exception several levels up, he must raise the same exception explicitly in the handler of each level. This exception propagation mechanism is called **explicit propagation.**

Ada [Ada95] declares exceptions by the statement exception. An exception not handled is automatically raised into the upper levels along the calling chain until a handler is found or until a program boundary is reached. Therefore, this propagation method is called **automatic or dynamic propagation**.

Example:

```
-- propagation.adb:  illustrate exception propagation

with Ada.Text_IO; use Ada;

procedure Propagation is

   E, F, G, H: exception;

```

```
  procedure A is
  begin
    Text_IO.Put_Line ("Begin A");
    -- ***********************************
    -- suppose an exception is raised here
    -- ***********************************
    Text_IO.Put_Line ("End A");
  end A;

  procedure B is
    procedure C is
    begin
      Text_IO.Put_Line ("Begin C");
      A;
      Text_IO.Put_Line ("End C");
    exception
      when E => Text_IO.Put_Line ("Caught E");
    end C;
  begin
    Text_IO.Put_Line ("Begin B");
    C;
    Text_IO.Put_Line ("End B");
  exception
    when F => Text_IO.Put_Line ("Caught F");
  end B;

begin
  Text_IO.Put_Line ("Begin Main");
  B;
  Text_IO.Put_Line ("End Main");
exception
  when G => Text_IO.Put_Line ("Caught G");
end Propagation
```

The program starts execution (main program) writing a line, and then calls procedure **B**. procedure **B** starts also writing a line, and then calls procedure **C**. Procedure C calls procedure A after writing a line. In procedure **A**, the program writes a line, and then an exception is thrown (a supposition). Because procedure **A** has not an exception handler, the exception is propagated into upper levels along the calling chain. The upper level in this case is procedure **C** which has declared an exception handler (for exception **E**). If the exception is not caught in procedure **C**, then the exception is reraised and propagated to procedure **B**. This procedure has a handler that will try to catch again the exception. The propagation can continue until a handler is found or until a program boundary is reached.

In C++ [Str91], there is no specific declaration for exceptions. User can raise an ordinary object as an exception by using the statement **throw.** A **try…catch** structure attaches handlers led by **catch** to a guarded block of code led by **try**. If the handler for a raised exception cannot be found locally, C++ unwinds the stack of the try block and propagates the exception to its caller. This procedure continues until a handler is found or until a default handler is called, which then aborts the program.

Java [GJS96] uses a mechanism similar to C++, adding the clause **finally** to the **try…catch** structure. The statements in **finally** are executed whether or not exceptions are raised.

Example:

```
class One_Exception extends Exception {
  int argument;
  public One_Exception (int i) { argument=i; }
}

```

```
class Another_Exception extends Exception {}

public class Try_Block {

  public static void main (String argv[]) {

    // Java "try" block with "catch" and "finally"

    try {

      // block of statements; may raise exceptions,
      // "break", "continue", or return.

      if (1==0) {
        throw new One_Exception (5);
      } else {
        throw new Another_Exception ();
      }

    } catch (One_Exception e) {
      // one handler
      System.out.println (e.argument);

    } catch (Another_Exception e) {
      // another handler
      e.printStackTrace (System.err);

    } finally {
      // final wishes; always executed no matter whether
      // we leave the block normally, with an exception,
      // because of a "break", "continue", or return
    }
  }
}
```

When the Java program enters in the **try** block, it tests the condition (1==0). In this case the condition is false, so the program will raise the exception **Another_Exception ()**. Inside the try block there are two exception handlers: one for exception **One_Exception** and another for **Another_Exception**. The first handler will fail to catch the exception that was raised, but the second handler will success, and it will execute the code of the handler (e.printStackTrace (System.err)).

SML [MTH90] like Java has exceptions that are themselves values. An exception name in Standard ML is a constructor of the built-in type **exn** [Pau91]. The exception declaration **exception exc_name** makes **exc_name** a new constructor of type **exn**. Raising an exception creates an **exception packet** containing a value of type **exn**. For example, **raise Ex** throws exception **Ex**. During evaluation, exception packets propagate under the call-by-value rule. If expression **E** returns an exception **packet** then that is the result of the application **f(E)** for any function **f**. An exception handler tests whether the result of an expression is an exception packet. SML uses the construct E **handle P1 => E1 | … | Pn => En** to define an exception handler [Paulson].

Example:

```
exception Neg
local
        fun search (n,i) = if n<0 then raise Neg else
                           i*i<=n
                           andalso (n mod i = 0 orelse search (n,i+1))
in
        fun composite n = search (n,2)
        fun prime n = not (composite n)
end;

(prime ~7) handle Neg => (print("The number is negative: \n");false)
```

The program finds whether a number is a prime number or not. It first declares an exception handler named **Neg** which is thrown when a negative number is passed to the function **search**. If the exception **Neg** is raised then the handler (last line of code) will catch it and its code (print("The number…")..) will be executed. Shall another kind of exception be raised in some of the three functions; the system will follow the calling chain (automatic propagation) to upper levels. Because there is no another exception handler, the exception will be considered an **uncaught exception** (the program terminates and returns to the operating system).

Handler binding

Handler binding attaches handlers to certain exceptions to catch their occurrences in the whole program or part of the program. There are three ways to bind handlers with exceptions [LS98]: **static**, where once a handler is attached to an exception, the same handler is used for every occurrence of that exception in the whole program or process; **semidynamic**, used by Ada, C++, and Modula-3, where different handlers associates with the exception in different context during an exception propagation; and **dynamic**, where different handlers can be attached to an exception in the same context.

Implementing exception handling

It is the process of receiving the notification, identifying the exception, and determining the association handler. There are several methods, which are divided into the following categories [LS98]:

- Stack unwinding. The handler defined first is checked first. If none can be found to handle the raised exception, the context stack is unwound, and the search begins within the new context. This is the method used in Ada, C++ and Modula 3.
- Handler pool. It is a handler chain, or lined list, or a table of handlers, each of which has been bound to a specific exception or group of exceptions. To find an associated handler, the pool is searched linearly.
- Combination of stack unwinding and handler pool. A separate handler chain is stored within the stack frame.
- Backtracking exception identifier bindings. It "backtracks" exception identifier bindings to determine a matching handler.
- Scanning instances of objects. It scans all the instances of an object for handler determination, since users can supply different handlers for the same exception raised in different instances.

3 A Model of CPS Translation and Interpretation

The middle part and key transformation in some functional language compilers is the conversion to CPS (continuation-passing style) language, which was defined and explained in chapter 2. We use CPS as our intermediate representation in our functional language compiler that was built for our experiments (figure 3.1).

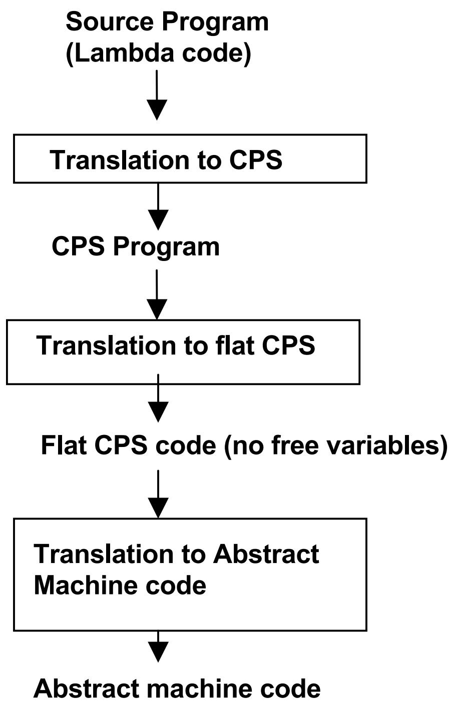

Figure 3.1 Overview of the compiler for the experiments

The CPS language is well-designed to match both the lambda calculus, which is the source language in our compiler, and the model of a von Neumann machine (represented by the abstract machine code). The compiler first translates lambda code into CPS expressions. Then, CPS expressions are translated into a free variables representation which is called Flat CPS. Flat CPS code consists of only one CPS function (no inner functions as in a normal CPS expression). Last, Flat CPS is translated into an abstract machine code.

We present in this chapter, a model of translation and execution that allows a programmer (or student/teacher) to write, translate, and execute programs in a source functional language (an extended lambda language) and a target CPS language. Both systems are based on the definitions of a semantic for CPS and a model of translation by Appel [App92]. The main contribution of our model is to collect everything (the model of translation and semantics) together into a working program and to create a whole framework which can be used to execute programs, allowing studying a wide range of performance assessments that can be discussed, highlighting the performance relationships among different elements. Figure 3.1, shows this model of translation-interpretation.

As we can observe in figure 3.2, a program written in an intermediate representation of a functional language like SML (in this case lambda code), is translated into a CPS program and then, evaluated using a specific input as data. After the evaluation, a value (the result) is obtained.

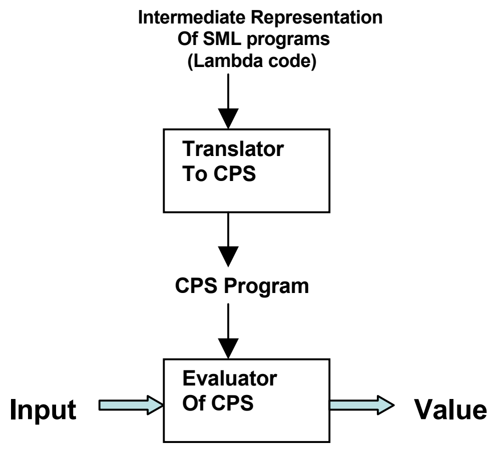

Figure 3.2 Model of translation-interpretation

3.1 A minicompiler for miniML

The first part of model described above, is a translator to CPS. This translator takes a program written in a lambda language (encoded into a tree-like data structure), and then makes a recursive traversal over the source-language program producing a CPS program.

The Lambda language

Figure 3.3 shows the definition of a lambda expression as an ML datatype.

```
type var=string

datatype lexp=
       VAR of var
      |INT of int
      |STRING of string
      |FN of var * lexp
      |FIX of var list * lexp list * lexp
      |APP of lexp * lexp
      |PLUS
      |SUB
      |MULT
      |LESS
      |EQ
      |MAKEREF
      |RAISE of lexp
      |HANDLE of lexp * lexp
      |COND of lexp * lexp * lexp  (* switch *)
```

Figure 3.3 Datatype for a lambda expression.

In this case, each value (a constructor) of type lexp can represent:

- A variable (VAR), an integer (INT), or a string (STRING);
- An anonymous (lambda) function (FN);
- A function declaration (FIX) where function names (var list) are bound to anonymous functions (lexp list) under the scope of a lambda expression;
- A function-calling construct (APP);
- A set of primitive operations for making arithmetic (PLUS, SUB, and MULT); comparisons (LESS, and EQ); and creation of references to memory (we use them when exceptions are declared).

- A primitive operation to evaluate an expression of type **exception** and to throw a user-defined or system exception (RAISE).
- A primitive operation **HANDLE** which evaluates the first argument, and if an exception is raised, then applies the second argument (handler) to the exception.
- A primitive operator **COND** used to test conditions **EQ** and **LESS**. Besides normal testing, this primitive is very important when a **HANDLE** tests for a determined exception.

Examples: The next table shows several examples of different lambda expressions using our notation. We also show, for clarity purposes, the corresponding code of the lambda expression in SML code.

SML	**LAMBDA**
1	**INT 1**
289 – (17 * 17)	**APP(SUB,RECORD [INT 289, APP(MULT,RECORD [INT 17,INT 17])])**
(fn x => x)	**FN ("x",VAR "x")**
(fn x => 3)	**FN ("x",INT 3)**
(fn x => 3) 9	**APP (FN ("x",INT 3), INT 9)**
(fn x => x) 9	**APP(FN ("x",VAR "x"),INT 9)**

if (3 = 5) then 2 else 7	COND(APP(EQ,RECORD[INT 3,INT 5]),INT2,INT 7)

if (2 < ((fn x => x) 3))
then 2
else ((fn x => x) 7)
	COND(APP(LESS,RECORD[INT 2,
	APP(FN("x",VAR "x"),INT 3)]),
	INT 2,APP(FN ("x",VAR "x"),INT 7))

let Fun fact(n)= if n<1 then 1 else n*(fact(n-1) in Fact(6) end	APP(FIX(["fact"], [FN("n", COND(APP(LESS,RECORD [VAR "n",INT 1]), INT 1, APP(MULT,RECORD[VAR "n", APP (VAR "fact", APP (SUB,RECORD [VAR "n",INT 1]))])))], VAR "fact"),INT 6)
“a string”	STRING "a string"
Exception Astring	APP (MAKEREF, STRING "Astring")
let fun f(n)=n*n in f(0) end	FIX(["f"], [FN("n",APP (MULT,RECORD [VAR "n",VAR "n"]))], APP(VAR "f",INT 0))

let
fun f(n)=n*n handle
ovfl=>0
in
f(1700)
end
	FIX (["f"],
	[FN ("n",
	HANDLE
	(APP (MULT,RECORD [VAR "n",VAR "n"]),
	FN("e",COND (APP (EQ,RECORD
	[VAR "e",VAR "ovfl"]),VAR "n",
	RAISE (VAR "e")))))],APP (VAR "f",INT 1700))

let fun g(x)=f(x) handle DIV=>2 fun f(y)= raise DIV handle MULT=>1 in g(2) end	FIX (["g","f"], [FN ("x", HANDLE (APP (VAR "f",VAR "x"), FN ("e", COND (APP (EQ,RECORD [VAR "e", APP (MAKEREF,STRING "DIV")]),INT 2, RAISE (VAR "e"))))), FN ("y", HANDLE (RAISE (APP (MAKEREF,STRING "DIV")), FN ("e", COND

	(APP (EQ,RECORD [VAR "e", APP (MAKEREF,STRING "MULT")]),INT 1, RAISE (VAR "e")))))], APP (VAR "g",INT 2))
let fun f(n)=n*n handle ovfl=>n fun run(x)= if x>1000 then f(17) else (run(x+f(17)-288)) in run(0) end	FIX (["f","run"], [FN ("n", HANDLE (APP (MULT,RECORD [VAR "n",VAR "n"]), FN ("e", COND (APP (EQ,RECORD [VAR "e",VAR "ovfl"]),VAR "n", RAISE (VAR "e"))))), FN ("x", COND (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 1700), APP (VAR "run", APP (PLUS, RECORD [VAR "x", APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])])))))], APP (VAR "run",INT 0))

Table 3.1 Examples of SML and Lambda expressions.

The CPS language

The CPS language used in our translator has three big differences with respect to those traditional compilers which use also CPS as an intermediate representation [App92]:

- Every function has a name.
- There is an operator for defining mutually recursive functions (instead of fixed point function).
- There are *n*-tuple primitive operators.

Besides that, we use the ML datatype declaration in order to prohibit ill-formed expressions. One important property of CPS is that every intermediate value of a computation is given a name. This makes easier the translation later, to any kind of machine code. For example the SML expression **289 – (17 * 17)** is translated to

```
PRIMOP(*,[INT 17,INT 17],["w2"],
          [PRIMOP (-,[INT 289,VAR "w2"],
                   ["w1"],[APP (VAR "k",[VAR "w1"])])])
```

in CPS notation, where **w1** and **w2** are intermediate names produced by the translator. We will explain in more detail later this example.

Another important aspect of CPS operations is that every argument is atomic; that means that only variables or constants are allowed to be arguments. The definition of a CPS expression as an ML datatype is shown in Figure 3.4.

```
datatype primop=
       gethdlr
      |sethdlr
      | +
      | -
      | *
      | <
      |equal
      |makeref
type var=string;
datatype value =
       VAR of var
      |INT of int
      |STRING of string
datatype cexp=
      |APP of value * value list
      |FIX of (var * var list * cexp) list * cexp
      |PRIMOP of primop * value list * var list * cexp
       list
```

Figure 3.4 Datatype for a CPS expression.

A primitive operator can be:

- gethdlr and sethdlr. Both are used for handling exceptions. The operator **gethdlr** obtains the current exception handler (or saving the old handler), and **sethdlr** updates the store with a current handler (re-install a new handler).
- +, -, *. Arithmetic operators for adding, subtracting, and multiplying two arguments.
- <, equal. Testing (comparison) operators for **less than** and **equal to**.
- makeref. This operator is used to create a reference (a pointer) to memory. We use **makeref** mainly to declare an exception.

A **value** datatype is defined as all the different kind of atomic arguments that can be used in a CPS operator. A value or argument can be a variable (**VAR**), an integer (**INT**), or a string constant (**STRING**).

Our CPS language has just three different kinds of expressions. They are:

- APP. It is used for calling a function (whose name is of type **value**), passing one or more arguments (using a list of values).
- FIX. As we mentioned before, in CPS all functions have a name. There are no anonymous functions. FIX is used to define a general-purpose mutually recursive function definition. The syntax of FIX defines a list of zero or more functions, with a name (type **var)**, arguments (type **var list)**, and bodies (type **cexp)**. All of these functions can be called (using the APP operator), from each body of the function or from the main body of the FIX expression (type cexp).
- PRIMOP. This stands for **primitive operator**. All primitives like handling exception, arithmetic, testing, and references, are built by using this constructor. The first field is the primitive name (primop type), the second and third fields are used for arguments and/or result names, and the fourth field is the continuation expression of the primitive operator.

A set of examples will clarify CPS notation. Next tables show the same examples from last table but including lambda code and corresponding CPS code.

Example # 1

```
INT  1
APP (VAR "k",[INT 1])
```

Where **APP(VAR “k”, [result])** is the initial continuation for any program in the CPS expression. This continuation is really what is called in functional programming, **the identity function (fn x => x)**.

Example # 2

```
APP(SUB,RECORD  [INT 289,
   APP(MULT,RECORD  [INT 17,INT 17])])
PRIMOP (*,[INT 17,INT 17],["w2"],
    [PRIMOP (-,[INT 289,VAR "w2"],["w1"],[APP (VAR "k",[VAR "w1"])])])
```

CPS evaluates first the multiplication operator, giving as a result **w2**, and then the continuation is evaluated (subtraction). At the end, the result is given to the initial continuation (**VAR w1**), which is also a continuation from the subtraction operation.

Example # 3

```
FN ("x",VAR "x")
FIX ([("F3",["x","k4"],APP (VAR "k4",[VAR "x"]))],APP (VAR "k",[VAR "F3"]))
```

A lambda function (anonymous or named function) corresponds to a FIX function, which uses a determined name. In this example, we can see that **F3** is the name assigned for the compiler to the function. Besides, this function has two arguments. The first one is variable **x** (same as lambda expression), and another one for **k4.** This is the continuation that takes the rest of the computation when the function is called from an application.

Example # 4

```
FN ("x",INT 3)
```

```
FIX ([("F5",["x","k6"],APP (VAR "k6",[INT 3]))],APP (VAR "k",[VAR "F5"]))
```

This example is very similar to the last one.

Example # 5

```
APP (FN ("x",INT 3), INT 9)
```

```
FIX
  ([("r7",["x8"],APP (VAR "k",[VAR "x8"]))],
   FIX
     ([("F9",["x","k10"],APP (VAR "k10",[INT 3]))],
      APP (VAR "F9",[INT 9,VAR "r7"])))
```

An anonymous function is applied a value (**INT 9**). Inside the body of the function, the bound variable is not used. So, the result will give just **INT 3**. In the CPS code we see two functions. The inner function corresponds to the anonymous function of the lambda expression. The outer function **r7** corresponds to the rest of the computation after the inner function has been evaluated. We can interpret **r7** as the normal return from the function. The CPS flow of execution starts calling function **F9**, which takes two arguments. **F9** then call **k10** (which takes the value **r7**), and last the identity function is evaluated.

Example # 6

```
COND(APP(EQ,RECORD[INT 3,INT 5]),INT2,INT 7)
```

```
FIX
   ([("F15",["z16"],
     PRIMOP
       (equal,[VAR "z16",INT 0],[],
        [APP (VAR "k",[INT 2]),APP (VAR "k",[INT 7])]))],
    PRIMOP
      (equal,[INT 3,INT 5],[],
       [APP (VAR "F15",[INT 0]),APP (VAR "F15",[INT 1])]))
```

A condition expression in lambda language produces a function in CPS language. The condition test for two arguments (3 and 5), and depending of the result of the test, make first or second options (2 or 7). With CPS, the evaluation start also testing the arguments, continuing with a call to **F15** with argument INT 0 if the result of the test was true, or a call to **F15** with argument INT 1 if it was false. The Function **F15** begins testing for the argument; if zero (true) then it finish with the identity continuation with argument INT 2 as a result. If not zero then the continuation is with argument 7.

Example # 7

```
APP(FIX(["fact"],
    [FN("n",
      COND(APP(LESS,RECORD [VAR "n",INT 1]),
      INT 1,
      APP(MULT,RECORD[VAR "n",
        APP (VAR "fact",
         APP (SUB,RECORD [VAR "n",INT 1]))])))],
    VAR "fact"),INT 6)
```

```
FIX
   ([("r29",["x30"],APP (VAR "k",[VAR "x30"]))],
    FIX
     ([("fact",["n","w31"],
       FIX
        ([("F32",["z33"],
          PRIMOP
           (equal,[VAR "z33",INT 0],[],
            [APP (VAR "w31",[INT 1]),
             FIX
              ([("r35",["x36"],
                PRIMOP
                 (*,[VAR "n",VAR "x36"],["w34"],
                  [APP (VAR "w31",[VAR "w34"])]))],
               PRIMOP
                (-,[VAR "n",INT 1],["w37"],
                 [APP (VAR "fact",[VAR "w37",VAR "r35"])]))]))],
         PRIMOP
          (<,[VAR "n",INT 1],[],
           [APP (VAR "F32",[INT 0]),APP (VAR "F32",[INT 1])])))],
      APP (VAR "fact",[INT 6,VAR "r29"])))
```

This example corresponds to the classical factorial function. The lambda code is built by using one FIX function (fact) which binds an anonymous function for the body of the factorial function. The factorial of 6 is evaluated. The CPS expression contains three FIX functions. One is for applying the identity function (**r29**); another one for the function factorial; and another one for the condition expression. The main difference in both programs is the way it accumulates the result. In the lambda code, the argument **n-1** and a return address are pushed into a stack and then the values and addresses are popped in order to get the factorial.

But with CPS we do not have return from calling function. The CPS code passes also the argument **n-1**, but instead of passing the return address, CPS passes a function (name) which contains the rest of the computation (see code in italic form). In this case, variable **w37** corresponds to the argument **n-1** passed to the factorial function, and variable **r35** is the function which corresponds to the rest of the computation. Function **r35** is an iterative function which computes the factorial by calling itself **n** number of times. In this case, the call to function **w31** is really to function **r35**, the value bound to **r31** (however in the last call **w31** has value **r29**, the initial argument passed in the first call to function **fact**, and the last function called in the program).

Example # 8

```
RAISE(APP(MAKEREF, STRING "except1"))
```

```
PRIMOP
   (makeref,[STRING "except1"],["w50"],
    [PRIMOP (gethdlr,[],["h49"],[APP (VAR "h49",[VAR "w50"])])])
```

The raise operator is used to throw an exception which is later caught or uncaught by a handler. In this example, we first create an exception named **except1**, which is thrown later. In CPS, the current exception handler is first returned (**gethdlr**), and then a jump to this handler is made using the declared exception (**w50**) as an argument.

Example # 9

```
FIX (["f"],
    [FN ("n",
         HANDLE
               (APP (MULT,RECORD [VAR "n",VAR "n"]),
                 FN("e",COND (APP (EQ,RECORD
                                 [VAR  "e",VAR "ovfl"]),VAR "n",
               RAISE (VAR "e")))))],APP (VAR "f",INT 1700))
```

```
FIX
   ([("f",["n","w55"],
      PRIMOP
        (gethdlr,[],["h56"],
         [FIX
            ([("k58",["x67"],APP (VAR "w55",[VAR "x67"])),
              ("n65",["e57"],
               PRIMOP
                 (sethdlr,[VAR "h56"],[],
                  [FIX
                     ([("F59",["e","k60"],
                        FIX
                          ([("F61",["z62"],
                             PRIMOP
                               (equal,[VAR "z62",INT 0],[],
                                [APP (VAR "k60",[VAR "n"]),
                                 PRIMOP
                                   (gethdlr,[],["h63"],
                                    [APP (VAR "h63",[VAR "e"])])]))],
                           PRIMOP
                             (equal,[VAR "e",VAR "ovfl"],[],
                              [APP (VAR "F61",[INT 0]),
                               APP (VAR "F61",[INT 1])])))],
                      APP (VAR "F59",[VAR "e57",VAR "k58"]))]))],
             PRIMOP
               (sethdlr,[VAR "n65"],[],
                [PRIMOP
                   (*,[VAR "n",VAR "n"],["w66"],
                    [PRIMOP
                       (sethdlr,[VAR "h56"],[],
                        [APP (VAR "k58",[VAR "w66"])])])]))]))],
    FIX
      ([("r68",["x69"],APP (VAR "k",[VAR "x69"]))],
       APP (VAR "f",[INT 1700,VAR "r68"])))
```

We will explain the lambda expression **handle** with more details. This expression has two parts. The first part is the expression that is going to be evaluated (multiplication expression). The second part is evaluated only if an exception is raised from the first part. The lambda code implements the second part using an anonymous function with one condition inside it. Whenever an exception is raised in the first part expression, the bound variable (**e**) of the function takes the exception name, and then the condition expression compares the bound variable (the exception) against a defined exception (**ovfl**). If the condition is true the handler catches the exception and continues with the first continuation (**VAR n**). If the condition is false it continues with second continuation (**raise e**).

As we explained before in this section, CPS implements exception handling by using two primitives: **gethdlr** and **sethdlr**. The first primitive getdlr executed (variable **h56**), saves the current handler in memory (at the end of the expression it will be restored). Next, primitive sethdlr with variable **n65** sets a new current handler (function n65). If the **multiplication** raises an exception (like overflow), a jump to the current handler (function **n65**) is performed. The first instruction to be executed in function **n65** is the restoration of the old current handler (sethdlr with variable h56). The rest of the code in function **n65** is the checking of the raised exception against exception overflow. At the end of the function a jump to function **k58** is made (this ends the execution of the handler). On the other hand, if the multiplication does not raise an exception, the next primitive sethdlr with variable **h56** restores the old current handler. Both cases (exception thrown or not), end jumping to function **k58**, which in turn jumps to the exit of the program: function **r68**. A more detailed description of implementing exception handling in the SML/NJ compiler is presented later in chapter 5.

The translator to CPS

The translation from a lambda expression to a corresponding CPS expression is made by a recursive traversal of the lambda expression. We saw in the last examples, that each lambda expression is represented in a hierarchical structure (a syntax tree) where each node represents an operation, and the children of a node represent the argument of the operation. For example, the tree for the lambda expression

```
FIX([“f”],
 [FN(“n”,APP (MULT,RECORD [VAR “n”,VAR “n”]))], APP(VAR “f”,INT 0))
```

is shown in figure 3.5.

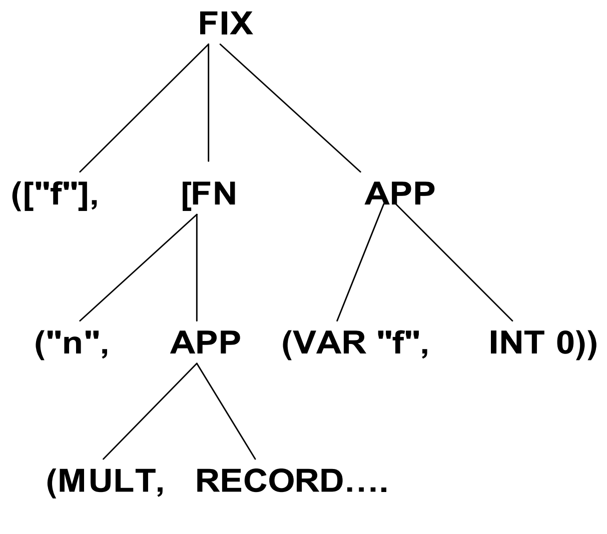

Figure 3.5 Syntax tree for a lambda expression.

We will describe the algorithm to convert any lambda expression to one in CPS. We do this by giving an ML function **f**, which transforms the ML data structure for lambda expressions given earlier. We also include a ML function

newVar: unit -> lexp

to create new variables. The function **f** and corresponding comments are shown in next table. We showed before several examples of the CPS translation.

```
local
   val count = ref 0;
   fun incr () = (count := !count + 1);
in
   type var = string;
   fun newVar (x) = (incr(); x^Int.toString (!count))
end
```

We start declaring a function to create new variables. That function uses a reference which is initialized with zero, and keeps increasing by one for each new variable. The new variables are created by concatenating a string of length one to a number (count).

```
fun f(lamb.VAR v, c) = c(VAR v)
|   f(lamb.INT i, c) = c(INT i)
|   f(lamb.STRING s,c) = c(STRING s)
```

To CPS convert a lambda variable, integer, or string, the continuation c is applied to the variable or constant. For example, in next function

f (lamb.INT 7,(fn x1=>APP (VAR "k",[x1])))

the continuation **c** is the second argument of function **f**, and it will produce

APP(VAR “k”,[INT 7])

```
|  f(lamb.HANDLE (A,B), c) = g (A,B,c)
```

Function g makes the translation of HANDLE. The explanation of the translation of primitive HANDLE is complex. In order to explain it with great detail, we will present next, the version written in the book of Appel [App92], which is a little easier to understand than the implementation.

```
f ( HANDLE (A,B),c) =
        PRIMOP(gethdlr,[],[h],
                FIX([(k,[x],c(VAR x)),
                    (n,[e], PRIMOP(sethdlr,[VAR h],[],[
                                f (B, λf.APP (f,[VAR e, VAR k]))]))],
                        PRIMOP(sethdlr,[VAR n],[],
                            [f (A, λv.PRIMOP(sethdlr,[VAR h],[],[APP(VAR k,[v])]))])))
```

A lambda HANDLE operator is translated into two mutually recursive functions, **k** and **n**, and a set of gethdlr and sethdlr CPS primitive operators inside and outside those functions. Function **n** will be the exception handler of the expression. Function **k** will apply continuation **c** (the continuation received by the whole expression), to the argument **x** (the result of the whole expression). This function will be called wherever or not an exception is raised (inside or outside the exception handler **n**). So, the flow of execution of this code will be:

- Start saving the current handler **h** (first gethdlr).
- Next, set the new handler n (sethdlr with variable n). It will be used only when an exception is raised in the first part of the expression).

- If an exception is raised in **A** (the code produced after translation of A), a jump to the new handler **n** is performed. Then, the handler will set the old current handler (sethdlr with variable **h**), and the code produced by the translation of **B** will be performed. In this code, a jump to function **k** will always be performed as the last operation of the function. This is because **k** is the continuation of **B**.
- If no exception is raised in **A** then there is no jump to the handler **n**, so a sethdlr of the old current handler **h** is executed, ending with a jump to function **k**.

```
| f(lamb.RAISE E, c) =
        let
           val h = newVar ("h")
        in
           f(E,(fn w=>cps.PRIMOP(cps.gethdlr,[],[h],
                        [cps.APP(cps.VAR h,[w])])))
        end
```

The code produced by the translator can be divided in two parts:

- There is some code produced from translation of **E**. This code is referenced by **w** in second part.
- The second part of the produced code, just gets the current handler h, and then jump to this handler passing **w** as an argument.

```
| f(lamb.FN (v,E), c) =
        let
           val F = newVar ("F");
           val k  = newVar ("k");
        in
           cps.FIX([(F,[v,k],f(E,(fn z=>cps.APP
                        (cps.VAR k,[z]))))],c(cps.VAR F))
        end
```

Two names of variables are needed. One is for the name of the CPS function, and another one for continuation k. The translator transforms a lambda function into a named FIX function. We know CPS functions do not return. Then a jump to continuation **k** is needed, taking **z** (The result of expression **E**) as an argument. Argument **v** has the same meaning in the CPS expression.

```
|  f(lamb.FIX(hx,bx,E),c) =
        let
           val w = newVar ("w")
           fun g(h1::h,lamb.FN(v,B)::b)=
                    (h1,[v,w], f(B, fn z=>
                         cps.APP(cps.VAR  w,[z]))::g(h,b)
           |   g(nil,nil) = nil
        in
           cps.FIX (g(hx,bx), f(E,c))
        end
```

Both types of FIX functions (lambda and CPS) are used for defining a set of named mutually recursive functions. Lambda function names and bodies are contained in two lists (hx, bx). Function **g** transforms both lists into a single list, containing function names, arguments, and bodies. The main expression E is also transformed with the current continuation **c**.

```
|  f(lamb.APP(lamb.MAKEREF,E),c) =
        let
           val w = newVar ("w")
        in
           f(E,fn v=>cps.PRIMOP(cps.makeref,[v],[w],
                          [c(cps.VAR w)]))
        end
```

This is an operator that takes one argument: the name of the exception. It is used to declare an exception. The operator does return a result, and continue in one way. **E** should be a string, which is later bound with the name **v**, to create a reference in the store. The result, a reference to the store, will be kept in **w**, which is then used in the continuation.

```
|  f(lamb.APP(lamb.PLUS,b.RECORD [x,y]),c) =
         convert_bin (cps.+, x, y,c)
|  f(lamb.APP(lamb.SUB,b.RECORD [x,y]),c) =
         convert_bin (cps.-, x, y,c)
|  f(lamb.APP(lamb.MULT,b.RECORD [x,y]),c) =
         convert_bin (cps.*, x, y,c)
```

Primitive arithmetic operators (PLUS, SUB, and MULT) are translated using the same format. As MAKEREF, they return one result, and continue in one way. Function **convert_bin** makes this transformation.

```
|  f(lamb.APP (lamb.LESS,b.RECORD [x,y]),c) =
         convert_jmp (cps.<,x,y,c)
```

Primitive operators for conditional branches (LESS and EQ) returns no result and continue in one of two ways. Function **convert_jmp** make this transformation.

```
|  f(lamb.APP (lamb.EQ,b.RECORD [x,y]),c) =
         convert_jmp (cps.equal,x,y,c)
```

```
|  f(lamb.APP (F,E), c) =
         let
            val r= newVar ("r");
            val x= newVar ("x");
         in
            cps.FIX([(r,[x],c(cps.VAR x))],
                 f(F,(fn f2=>f(E,(fn e=>cps.APP(f2,
                                    [e,cps.VAR r]))))))
         end
```

CPS functions do not have returns. Then, if we want to translate a lambda function call, we need to create a continuation function (it will replace the return address). This function is named **r**. We also need to evaluate **F** and **E**, from which **f2** and **e**, will refer to these values. Next, a jump to **f2** using **e** as the first argument and **r** (the continuation) as the second will be applied.

```
|  f(lamb.COND (test,exp1,exp2),c) =
        let
           val fname= newVar ("F")
           val z= newVar ("z")
           val E1= f (exp1, c)
           val E2= f (exp2, c)
           val f2= (fname, [z], cps.PRIMOP (cps.equal,
                   [cps.VAR z,cps.INT 0],[],[E1,E2]))
        in
           cps.FIX ([f2], f (test, (fn v=>cps.APP
                                  (cps.VAR fname,[v]))))
        end
```

For the primitive condition COND we need to create a FIX function. In the body of the recursive function there is a primitive operator for conditional branch (equal), that test if the argument of the function is zero. The main expression of the FIX operator is the translated code for the test.

```
and
  g(A,B,c1)=
        let
           val h= newVar ("h")
           val e= newVar ("e")
           val k= newVar ("k")
            val n= newVar ("n")

           val seth1=
            cps.PRIMOP(cps.sethdlr,[cps.VAR h],
              [],[f(B,(fn f2=>cps.APP(f2,[cps.VAR e,
                                  cps.VAR k])))])
           val seth2=
            cps.PRIMOP(cps.sethdlr,[cps.VAR n],
              [],[f(A,(fn v=>cps.PRIMOP(cps.sethdlr,
                [cps.VAR h],[],[cps.APP(cps.VAR k,
                                          [v])]))))])
           val x= newVar ("x")
           val fix1=
            cps.FIX([(k,[x],c1(cps.VAR x)),(n,
                                     [e], seth1 )], seth2 )
        in
          cps.PRIMOP(cps.gethdlr,[],[h],[fix1])
        end
```

Function g implements lambda operator HANDLE. First, four new variables are created for the names of the functions (**n** and **k**), the handler **h**, and an argument **e**. Then, **seth1** contains the code of the body for new handler **n**, and **seth2** the code for the main expression of FIX. Name **fix1** contains the code of the entire FIX expression, including seth1 and seth2. And finally, The whole code for HANDLE is contained in cps.PRIMOP(cps.gethdlr,[],[h],[fix1]).

```
and
   convert_bin (bin_op, x, y, c) =
        let
             val w= newVar ("w")
             fun c2 vx vy =
                cps.PRIMOP (bin_op, [vx,vy],[w],
                                   [c (cps.VAR w)])
        in
             f (x, (fn xv => f (y, c2 xv)))
        end
```

This function converts primitive arithmetic operators PLUS, SUB, and MULT. We need first to convert the argument expressions of the primitive, which are always two (**x** and **y**). Conversion of **y** is made first, taking a PRIMOP operator as a continuation. Then, conversion of **x** is performed taking also a PRIMOP operator as continuation. So, the result will give a primitive operator for **x** inside another primitive operator for **y**. The last continuation is always the initial **c** continuation applied to the last result **w**.

```
and
   convert_jmp (jmp_op, x, y, c) =
        let
           val w=newVar ("w")
           fun c2 vx vy = cps.PRIMOP (jmp_op, [vx,vy],
                      [], [c (cps.INT 0), c (cps.INT 1)])
        in
           f (x, (fn xv => f (y, c2 xv)))
        end
```

Observing the code produced by this function, we find only two differences with respect to the code for function **convert_bin**. The CPS primitive operators produced have not result (the third field is the empty list), and there are two possible continuations for that primitive. These continuations take as arguments **INT 0** or **INT 1** which represent true or false respectively.

3.2 A conceptual and executable framework

The semantic of CPS is described by Appel [App92] in chapter 3, where he explains the meaning of CPS expressions by using denotational or continuation semantics. This semantics is defined as a functor in SML. The functor takes a CPS structure, a datatype for the values allowed in the semantics, and some data definitions as arguments. Then, a function evaluates a CPS expression, using an empty environment and store at the beginning of the evaluation.

We implemented this continuation semantic by defining some needed functions, an initial environment, and a store. Also, we linked some other programs like the translator of CPS, and together we had as a result a **conceptual and executable framework of functional programming** (see figure 3.6).

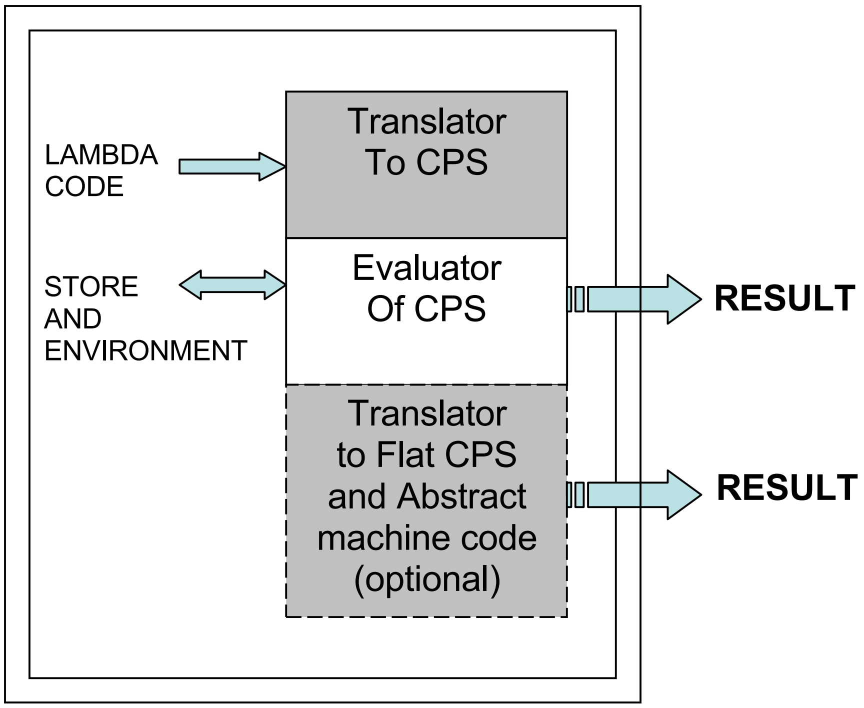

Figure 3.6 Conceptual and executable framework.

This conceptual framework focuses on the experience of learning the CPS concepts by using a framework of CPS programming. It can serve as an aid in gaining a coherent understanding of the CPS programming. The most important element of this framework is that programs in lambda and CPS code can be directly compiled and executed and, the programmer can see how this source code (lambda) is transformed into correspondent CPS code together with important components of the framework, like an environment and a store. From this framework a teacher and/or student should be able to write their own programs, test them with different data, make experiments, etc. Research based upon the experiment approach can be conducted in order to study the different structures to use in a program, and so to determine what the best approach is.

Evaluator of CPS

The evaluator of CPS is a program which takes a CPS program, an input, and performs an evaluation or execution of the CPS program, giving as a result a denotable value (which is later converted into a string). The input is formed by two components:

- An environment. This is a function that maps CPS variables to denotable values (result values). The initial environment of the evaluator is created by three functions:

  ```
  val env0 = fn v=>raise Undefined (v)
  val env1 = bind(env0,"k", FUNC ic)
  val env2 = bind (env1, "ovfl", overflow_exn)
  ```

  The first value bound to the environment is the empty environment (raise Undefined); the second value is the initial continuation (identity continuation); and the third value is a predefined exception (overflow).

- A store. A function that maps locations (addresses) to denotable values. The initial store of the evaluator are three functions:

  ```
  val store0 = (100, fn l=>raise Exc_Overflow l,fn _=>0)
  val store1 = upd(store0,handler_ref,FUNC default_handler)
  val store2 = upd(store1, overflow_loc, STRING "-overflow-")
  ```

Store0 establish that the first unused location is address 100. Addresses before 100 are used for keeping values like system exceptions. This store is the empty location (a raise to an exception); the second location has the initial default handler; and the third location has the predefined overflow exception handler.

The output or result is a denotable value. It can be any of these values:

- INT. It denotes the type integer.
- FUNC. A constructor of function type. It takes a list of denotable values, and a store, yielding an answer (a string).
- STRING. It denotes the string.
- ARRAY. An array of locations. Our implementation uses it to store references to exceptions.

In next tables, we show the implementation and some comments of the evaluator of CPS.

```
type nextloc= loc -> loc
fun nextloc (l)=l+1
type answer = string
type var = string
datatype dvalue =
          INT of int
          |FUNC of dvalue list ->
```

```
                    (loc*(loc->dvalue)*(loc->int))->
                    answer
            |STRING of string
            |ARRAY of loc list
```

Function **nextloc** is used to generate new locations in the store. **Answer** is the result of all the execution of a program. The datatype dvalue define a set of constructors representing the denotable values of the semantic. Denotable Values can be used as arguments, variables, etc. They can be an integer, a string, an array or a function. **ARRAY** values are a list of type **loc** (integers) and they are a mutable data structure (they can be modified using the upd function). We use dvalues of type **ARRAY** to store references, used when an exception is declared. A dvalue of type **function** takes a list of actual dvalues and a store.

```
type store = loc * (loc -> dvalue) * (loc -> int)
type handler_ref= loc
val overflow_loc = 7;
val overflow_exn: dvalue = ARRAY [overflow_loc]
```

The type of the store is loc*(loc->dvalue)*(loc->int), where **loc** represent the next unused location, (loc->dvalue) a mapping from locations to dvalues, and (loc->int) a mapping from locations to integers. The **current handler** is kept in a special location in store. We decided to store the address (reference) of the overflow exception in the element 7 of a dvalue ARRAY.

```
fun upd ((n,f,g):store, l: loc, v: dvalue) =
                    (n, fn i => if i=l then  v  else f i, g)
```

Function **upd** is used to modify the store, given a location and its value. We use upd every time a new exception handler is set by the operator sethdlr, or when a new reference is created by the operator makeref (remember we use it to create new exceptions).

```
fun fetch ((_,f,_): store) (l: loc) =   f l
```

Function **fetch** is used for getting a value (denotable value) from store using a determined location.

```
exception Undefined of var and Exc_Overflow of loc
```

We define two exceptions: Undefined that is used when a value is not in the environment (undefined variable), and Exc_Overflow when a value is not in the store.

```
fun do_raise exn s =
                let val FUNC f= fetch s handler_ref in f [exn] s end
```

Function do_raise catches overflow exceptions for arithmetic operations. It can be though as a system exception handling for the CPS. The function uses the default handler, which is bound in the store with location **handler_ref**, and then passes parameter **exn** to this default handler.

```
fun overflow(n:unit->int, c:dvalue list ->store->answer)=
        if (n() >=minint andalso n() <=maxint)
                handle Overflow=>false
        then  c [INT(n())]
        else  do_raise overflow_exn
```

Function overflow checks for limit (minimum and maximum) in results of arithmetic operations. There is a handle expression which catches SML overflow exceptions (in the metalanguage). The function calls **do_raise** function if there is a violation of the limits allowed in the program. If there is not overflow, then the result of the arithmetic operation is passed to the continuation **c**.

```
exception bad_equality and Error
```

Two exceptions are defined: **bad_equality** is raised when two non compatible denotable values are compared; **Error** is raised when the result of the program is not a denotable value.

```
fun evalprim (a.gethdlr, [], [c]) =
    (fn s => c [fetch s handler_ref] s)
|   evalprim (a.sethdlr, [h], [c]) =
        (fn s => c [] (upd(s,handler_ref,h)))
|   evalprim (a.+,[INT i, INT j],[c]) =
        overflow(fn ()=> (i + j),c)
|   evalprim (a.-,[INT i, INT j],[c]) =
        overflow(fn ()=> (i - j),c)
|   evalprim (a.*,[INT i, INT j],[c]) =
        overflow(fn ()=> (i * j),c)
|   evalprim (a.<,[INT i, INT j],[t,f]) =
        if i<j then t[] else f[]
|   evalprim (a.equal,[INT i, INT j],[t,f]) =
        if i=j then t[] else f[]
|   evalprim (a.equal,[ARRAY [i],ARRAY [j]], [t,f]) =
        if i=j then t[] else f[]
|   evalprim (a.makeref,[v],[c])=
        (fn (l,f,r)=>c [ARRAY [l]] (upd ((nextloc l,f,r),l,v)))
|   evalprim (a.equal, [_,_], [t,f]) = raise bad_equality
```

Function evalprim evaluates a primitive operator (PRIMOP) applied to arguments. The first two primitive operators: **gethdlr** and **sethdlr** are used for exception handling. A gethdlr operator fetches the current exception handler (handler_ref) from the store. A sethdlr operator sets (updates) a new current handler in the store. Integer addition, subtraction, and multiplication just make the computation and if there is no overflow, applies **c** to the result. Integer comparison just tests two integers, and depending of the result, it applies one of two continuations (for true or false) to the empty list. Primitive makeref is important in exception declaration. The operator first inserts the denotable value **v** which can be the name of the exception, in the first available location in the store. Then, it inserts the store location where **v** was saved in the environment.

```
type env = a.var -> dvalue
```

The type of the environment is a function from a variable (string) to a denotable value.

```
fun V env (a.INT i) = INT i
|   V env (a.STRING s) = STRING s
|   V env (a.VAR v) = env v
```

Fun **V** converts CPS values to denotable values. For variables the function has to lock up the environment.

```
fun bind (env:env, v:a.var, d) =
          fn w => if v=w then  d
                else  env w
```

Function bind produces a new environment (a function of type a.var -> dvalue) by binding a new variable with a denotable value.

```
fun E (a.APP(f,vl)) env =
        let val FUNC g = V env f
        in g   (map (V env) vl)
        end
```

Function **E** is the function which takes the whole CPS expression, an environment, and a store, and then it evaluates the expression giving as a result a value of type answer (a string). Function application first locks up for function f in the current environment; this gives as a result a function which is applied to a set of arguments obtained (converted) from the environment.

```
|  E (a.PRIMOP(p,vl,wl,el)) env =
        evalprim(p,
                map (V env) vl,
                map (fn e => fn al =>
                        E e (bindn(env,wl,al)))
                        el )
```

In order to evaluate a primitive operator, we first convert the arguments using the current environment (map (V env) vl). Then the continuation (a function) of the evalprim function is built by using the continuation of this function (E), and a new environment with the addition of element **wl**.

```
|  E (a.FIX(fl,e)) env =
                let fun h r1 (f,vl,b) =
                        FUNC(fn al => E b (bindn(g r1,vl,al)))
                and g r = bindn(r, map #1 fl, map (h r) fl)
                in E e (g env)
                end
```

Function E with a mutually recursive function **FIX (fl,e)** evaluates expression **e** in the augmented environment **g**. The augmented environment is built by binding the list of recursive function names (map #1 fl), with the list of bodies (map (h r) fl) of each of these recursive functions (FUNC(fn al => E b (bindn(g r1,vl,al)))) , and using theirs respective local variables (bindn(g r1,vl,al)).

```
fun ic [INT i] _    = Int.toString i
|   ic [STRING s] _ = s
|   ic [FUNC _] _   = "fn"
|   ic [ARRAY [l]] _= "ref "^(Int.toString l)
|   ic _           _= raise Error
```

This function (ic), is used to produce **answer** (an string) as a result of the evaluation. The function just transforms a denotable value to a string.

```
fun default_handler [ARRAY [l]] s =
  let
    val STRING e =fetch s l
  in
    "EXCEPTION "^e
  end
|   default_handler [_,ARRAY [l]] s =
  let
    val STRING e =fetch s l
  in
   "EXCEPTION "^e
  end
```

This function allows the program to display the output EXCEPTION name_exception whenever a user defined exception is raised. Remember that the name of the exception (string) is saved in memory, maintaining the location **l** in a denotable value of type ARRAY. So, using **l** as the location and the **fetch** function we can access the name of the raised exception.

```
val env0 = fn v=>raise Undefined (v)
val env1 = bind(env0,"k", FUNC ic)
val env2 = bind (env1, "ovfl", overflow_exn);

val store0 = (100, fn l=>raise Exc_Overflow l,fn _=>0)
val store1 = upd(store0,handler_ref,FUNC default_handler)
val store2 = upd(store1, overflow_loc, STRING "-overflow-")
```

We initialize the environment with three new bindings (fn v=>raise Undefined (v), FUNC ic, and overflow_exn), and the store with three new store locations (fn l=>raise Exc_Overflow, FUNC default_handler, STRING "-overflow-"), where the last one is the handler for the overflow exception.

```
fun eval (vl,e) dl = E e env2 store2
```

Finally, function eval will take a CPS expression **e**, two lists of variables and denotable values (**vl** and **dl**), and it will call function **E** passing formal parameters **e**, **env2**, and **store2**.

4 The Abstract Machine

Continuation-passing style is the representation that we use as intermediate code because it is closely related to Church's lambda calculus and to the model of von Neumann, represented by our target abstract machine language (see figure 3.1). Each operator of CPS corresponds to one operator in our target abstract machine code. In order to test the performance of the CPS code we implemented an abstract machine.

The machine has an instruction set, a register set and a model of memory, and executes programs written in abstract machine code. Figure 4.1 illustrates the components of the abstract machine.

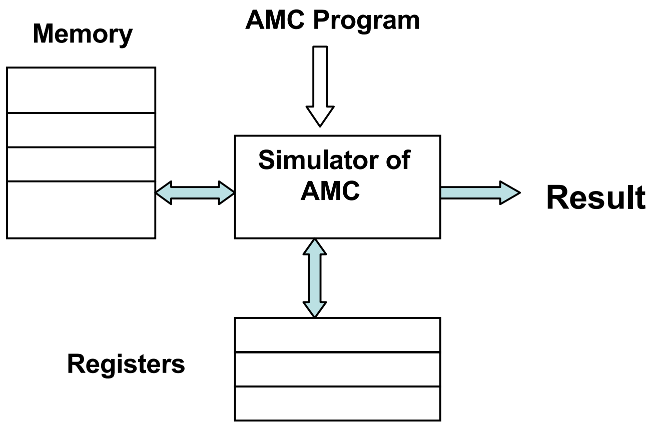

Figure 4.1 Components of the Abstract Machine

4.1 A generator of abstract machine code (AMC)

Flat CPS is in a form which is easily translated to abstract machine code (see figure 3.1). The abstract machine is modeled after a conventional von Neumann machine. The AMC is essentially an assembly-language program, and like any abstract machine it has some advantages with respect to a real machine: first, performance analysis is easier, and second it is easier to simulate.

The abstract machine language

Figure 4.2 shows the definition of the abstract machine instructions as an ML datatype.

```
datatype instruction =
           LABEL of string
          |JUMP of string
          |CJUMP of relop * exp * exp * string * string
          |LOAD of exp * exp
          |STORE of exp * exp
          |ADD of exp * exp * exp
          |SUB of exp * exp * exp
          |MUL of exp * exp * exp

and     exp=
           MEM of string
          |NAME of string
          |CONST of int
          |STRING of string
          |REG of int

and     relop=   EQ | LT
```

Figure 4.2 Datatype for an abstract machine instruction

Where a data or expression **exp** can be:

- An address of memory represented by a name (string) of a register, variable, etc.
- The name of a label, which represents an address.
- A constant for an integer data.
- A string data.
- The number (integer) of a register.

And an abstract machine **instruction** can be:

- A label which is really not an instruction, but just an address. Whenever the simulator finds a label it just increases the program pointer, in order to read the next instruction.
- A jump instruction is an unconditional branch to a label.
- A CJUMP is a conditional jump to one of two labels depending of the result of the test.
- A load or move from memory into a register.
- A store from a register or a string into a memory address.
- Arithmetic operations to add, subtract, or multiply two values, producing a result which is stored into memory.

We illustrate the abstract machine code with a complete program in SML, Lambda, CPS, flat CPS, and abstract machine code.

SML

```
let
        fun f(x)= x*5
in
        f(4)
end
```

LAMBDA

```
FIX(["f"], [FN ("x",
                APP(b.MULT,RECORD [VAR "x",INT 5]))],
          APP(VAR "f",INT 4))
```

CPS

```
FIX
  ([("f",["x","w1"],
     PRIMOP (*,[VAR "x",INT 5],["w2"],[APP (VAR "w1",[VAR "w2"])]))],
   FIX
     ([("r3",["x4"],APP (VAR "initialNormalCont",[VAR "x4"]))],
      APP (VAR "f",[INT 4,VAR "r3"])))
```

FLAT CPS

```
FIX
  ([("f",["x","w1"],
     PRIMOP (*,[VAR "x",INT 5],["w2"],[APP (VAR "w1",[VAR "w2"])])),
    ("r3",["x4"],APP (VAR "initialNormalCont",[VAR "x4"]))],
   APP (VAR "f",[INT 4,VAR "r3"]))
```

AMC

```
0       LOAD Const 4,Reg 1
1       LOAD Mem r3,Reg 2
2       JUMP  Name f
3    LAB f:
4       STORE Reg 1,Mem x
5       STORE Reg 2,Mem w1
6       MUL    Mem x,Const 5,Mem w2
7       LOAD Mem w2,Reg 1
```

```
8           JUMP   Mem w1
9      LAB r3:
10          STORE            Reg 1,Mem x4
11          LOAD Mem x4,Reg 1
12          JUMP   Mem initialNormalCont
13     LAB end:
```

We can see the different representations of the program after each phase of the compilation process, especially the last one: the abstract machine code. The code in the AMC performs the following operations:

- Instructions 0 and 1 pass the parameters in registers 1 and 2.
- Instruction 2 is a jump to label **f**.
- Instructions 4 and 5 store the parameters in memory.
- Instruction 6 multiplies first parameter (constant 4) by constant 5.
- Instruction 7 passes as a parameter the result of the multiplication in register 1.
- Instruction 8 is a jump to address r3 (the value of variable w1).
- Instruction 10 stores the parameter into memory address x4.
- Instruction 11 passes the value of x4 into register 1. This register always keeps the final result.

- Instruction 12 jumps to the initial continuation **initialNormalCont,** a fixed address or constant in memory that represents the end of any program (in the first CPS example of section 3.1 we explained the meaning of the initial continuation in a CPS program).

4.2 A simulator for the AMC

The simulator is a program which emulates a real computer. It is a piece of software that runs an AMC program. In order to emulate a real computer it uses three data structures which mimic a memory for data values, a memory for code, and a set of registers (see figure 4.1). It also uses two variables that keep the current program pointer (PC) for the code, and the current stack pointer (SP) for the data. The main routine of the simulator is a recursive function that keeps reading instructions from the AMC program. Next, we describe the algorithm that carries out the simulation of an AMC program.

Input: An AMC program (list of instructions).

Output: A value or result after executing the AMC program.

Method:

- Convert the list (AMC program) into an array (more convenient for the simulation)
- Initialize PC and memory pointers with initial address. PC points to first AMC instruction and memory pointer to address zero in memory.

- Start main function which keeps reading instructions pointed by PC, executing the operations (storing, loading, jumping, adding, etc.), and updating the value of PC.

Example: We now show the execution by the simulator of the AMC program shown in the last section. We display different stages of execution with the respective values in memory and registers. Memory and register values are shown before the displayed instruction is executed. As you can observe in the example, the memory of the abstract machine is an array of tuples, where the left component of the tuple is used for names or variables and the second for the value assigned to such names or variables.

INSTRUCTION 0: LOAD Const 4,Reg 1

```
MEMORY =
 [|("ovfl","ovfl"),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
  ("",""),("","")|]

REG1 = ""
REG2 = ""
```

At the beginning memory only contains the value of a pre-defined exception (overflow). Registers 1 and 2 are empty.

INSTRUCTION 2: JUMP Name f

```
MEMORY =
  [|("ovfl","ovfl"),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("","")|]

REG1 = "4"
REG2 = "r3"
```

Before executing instruction 2, registers 1 and 2 already contain the values passed as parameters.

INSTRUCTION 8: JUMP Mem w1

```
MEMORY =
  [|("ovfl","ovfl"),("x","4"),("w1","r3"),("w2","20"),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("","")|]

REG1 = "20"
REG2 = "r3"
```

Before executing instruction 8, the variables **x**, **w1**, and **w2** contain values 4, r3, and 20 respectively as a result of instructions 4-6, and register 1 contain value 20 as a result of instruction 7.

INSTRUCTION 8: JUMP Mem initialNormalCont

```
MEMORY =
  [|("ovfl","ovfl"),("x","4"),("w1","r3"),("w2","20"),("x4","20"),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),("",""),
   ("",""),("",""),("",""),("","") |]

REG1 = "20"
REG2 = "r3"
```

At the end of the program, register 1 contain the result of the multiplication (instruction 11 assigned it). In memory variable x4 get the same value which was passed as a parameter in instruction 10.

5 Exception-Handling Overhead

This chapter considers the implementation of exception handling. We examine programs written in the functional language ML. We tested two compilers: SML/NJ version 110.0.7 and OCAML version 3.06. We show that these programs with exception handling have runtime overhead even when no exceptions are thrown. Last, we describe the source of the overhead in programs compiled with the SML/NJ compiler.

5.1 Introduction

Many modern programming languages, for example, Java, Modula-3, and SML, provide mechanisms for dealing with exceptional conditions detected by hardware or software. They also allow the programmer to define other unusual events and use the same mechanisms to deal with them when they arise. These mechanisms are collectively called exception handling.

An exception can be defined as a condition brought to the attention of the operation's invoker, which becomes part of normal exit or return; or as an error or an event that occurs unexpectedly or infrequently, and includes an error or a signal. The exception handler is the code attached to an entity (program, procedure, object, expression, etc) that is executed when an exception occurs

[LS98]. A handler may catch one or more kinds of exceptions and may fail to handle other kinds of exceptions.

Though the syntax is different, exception handling in most modern programming languages is pretty much the same. In Ada [Ada95] and Modula-3 [Nel91] exceptions are declared using the keyword **exception**. The **raise** statement raises an exception. A **begin … exception … end** construct is used to associate handlers to some block of code in Ada. In C++ [Str91] there is no specific declaration for an exception. Users can raise an ordinary object as an exception by using the statement **throw**. A **try{...} catch{…}** structure attaches handlers led by the keyword **catch** to a guarded block of code led by the keyword **try**. Modula 3 uses the construct **try … except … end** and **try … finally … end** that are used to bind handlers to a code block and to clean up resources. The code led by the keyword **finally** is executed whether or not exceptions are raised. One interesting feature of Java [GJS96] is that it throws objects that are instances of the predefined class Throwable.

Some functional languages have exception handling. Some do not. Lazy functional languages like Haskell [Hud90], try to eliminate dependencies on the order of evaluation. Some exception handling introduces such situations. For example in the expression **(raise x, raise y)** the order of evaluation depends of which exception is first raised**.** Consequently, these languages do not have exception handling mechanisms.

Eager functional languages, on the other hand, sometimes have exception handling mechanisms like the imperative languages. In ML [MTH90], exceptions are declared with the keyword **exception**, and raised with the keyword **raise**. An exception can be handled with the construct **<expression> handle <match>** where match is a set of patterns (P) of the type P_1 =>E_1 | … | P_n => E_n and **E** is an expression (see SML example in chapter 2)**.** CAML [Ler00] uses a similar syntax. When declaring an exception, it uses the keyword **exception** too. The keyword r**aise** is used for throwing an exception, and **try <expression> with <match>** for handling the exception.

5.2 Implementation of exceptions (SML/NJ)

In chapter 3 we showed an example of translating a **HANDLE** and **RAISE** lambda expression into a corresponding CPS expression. We also described the algorithm to convert any lambda expression (like HANDLE and RAISE) into a CPS expression. In this section we give more details about the implementation of exception handling expressions in lambda and CPS language.

The SML/NJ [AM91, App92, and AT89] compiler translates a source program into a machine-language program in several phases. The first phase produces an abstract syntax tree. The second phase transforms this tree into lambda expressions. Then, the third phase translates the lambda code into continuation passing style (CPS) code, which is later converted into a no nesting function code (flattened CPS code). Last, many optimizations are performed, and machine code is produced. The run-time system uses a heap instead a stack. The absence of function return (a call-with-continuation instruction does not return like a normal function) means that a run-time stack is not required to execute programs. SML/NJ keeps all the activation records (closures) on the garbage-collected heap.

We study a simplified version of the SML programming language represented by the data type *lexp* (lambda expression)

```
datatype lexp = VAR of lexp                    |
                FN of var * lexp               |
                      ......
                RAISE of lexp                  |
                HANDLE of lexp * lexp
```

Where **RAISE** and **HANDLE** are the kind of lambda expressions for exception handling in the lambda language of the compiler. An exception handler in ML is a set of patterns **P** of the type P_1 =>E_1 | … | P_n => E_n where **P** is a pattern, usually an exception name, and E is a given expression. So the exception handler resembles a case construct. In the lambda language, a handler is just a function taking an expression *exn* as an argument. **RAISE** evaluates an expression of type *exn* and then raises that exception. **HANDLE** evaluates its first argument, and if an exception occurs it applies the second argument to that exception. The second argument is an expression of type *exn* $\rightarrow$ A, where A is the type of the first argument. To implement exception handlers, there is a distinguished location in the store, containing the current exception handler; each exception handler is just a continuation taking an *exn* argument. A **HANDLE** just installs a new exception handler upon entry, and re-installs the previous handler upon exit. A **RAISE** just passes its argument to the current handler.

In the third phase of the compilation, and after some optimizations and representation decisions have been made, the lambda code is translated into CPS code.

The CPS primitive operators used by the SML/NJ compiler for exceptions are **gethdlr** (get the current exception handler) and **sethdlr** (update the store with a new exception handler). A complete explanation of gethdlr and sethdlr operators with some examples was given in chapter 3. The store has a special location (a special register) in which the "current exception handler" is kept. This is a function, which is called in order to "raise" an exception. Primitive operators and CPS expressions are described in ML as follows:

```
datatype primop = gethdlr | sethdlr | makeref | * | - | + | < | …
datatype cexp =    PRIMOP of primop * value list
                                     * var list * cexp list
                  |FIX of (var * var list * cexp)list * cexp
                  |APP of value * value list
```

The primitive operators **gethdlr** and **sethdlr**, take 2 arguments, return no result, and continues in only one way. This type of operator is executed only for the side effect on the store. The semantics of the primitive operators **gethdlr** and **sethdlr** are:

Evalprim (gethdlr, [], [c]) = (fn s => c [fetch s handler_ref] s)

Evalprim (sethdlr, [h], [c]) = (fn s => c [] (upd(s, handler_ref,h)))

Where **fetch** is a function for obtaining a value (the current exception handler) from a location in the value store **s**; **upd** is a function for updating a location with a new value **h**, producing a new store **s**; **handler_ref** is the address where the actual or current exception handler is stored in the store; **h** is the exception handler to fetch or to set; and **c** is the current continuation. So, in the first operator a fetch for recovering the current exception handler to the store **s** in location **handler_ref** is performed. In the second operator, an update of the store **s** in location **handler_ref** with the value of exception handler **h** is made.

A **function** *convert* (or function **f** like in chapter 3) performs the translation. We show again the case of *convert* for lambda expressions **HANDLE** and **RAISE** (it was also described in section 3.2). The function *convert* takes two arguments: a lambda-language expression *lexp* and a continuation function *c*. The result is a CPS expression *cexp*: the original lambda expression, converted to CPS.

```
convert( HANDLE (A,B),c) =
     PRIMOP(gethdlr,[],[h],
      FIX([(k,[x],c(VAR x)),
          (n,[e], PRIMOP(sethdlr,[VAR h],[],[
             convert(B, λ f.APP (f,[VAR e, VAR k]))]))],
       PRIMOP(sethdlr,[VAR n],[],
        [convert(A,λv.PRIMOP(sethdlr,[VAR h],[],[APP(VAR k,[v])]))])))
```

The translation of **HANDLE** must first save the old handler **h**. Then, it makes a continuation **k** corresponding to the context of the entire **handle** expression. Next, it makes and installs a new exception handler **n**. Finally, the first operand A of the **handle** expression is executed, with a continuation that re-installs **h** and then invokes **k**.

The new handler **n**, if invoked, first re-installs **h** and then evaluates the second operand B of the **HANDLE** expression, continuing with **k**.

Raising an exception is much simpler:

```
convert (RAISE(E), c) = convert(E,
        λw.PRIMOP(gethdlr,[],[h], [APP(VAR h,   [w])]))
```

The function first evaluates the exception value **E**, yielding a value referred to by metavariable **w**. Then the current handler **h** is extracted and applied to **w**. In other terms, we first extract the current exception handler **h** from the store (at a specific location) and then apply it to the expression E. The continuation **c** is ignored completely because raising an exception disrupts the normal flow of control. This is the way the SML/NJ compiler translates exception handling.

5.3 Overhead in Exception Handling of SML/NJ

Implementing exception handling in SML/NJ produces runtime overhead. In order to experimentally verify this, we wrote two small programs in SML.

Program 1: Exception Handler

```
val t=Timer.startRealTimer();
local
        fun f(n)= n*n handle Overflow=>n
in
        fun run(m) =    if m > 1000000 then f(17)
                        else (f(17) ; run(m+1))
end;
run(0);
val ct=Time.toReal(Timer.checkRealTimer t);
```

Program 2: No Exception Handler

```
val t=Timer.startRealTimer();
local
        fun f(n)= n*n
in
        fun run(m) = if m > 1000000 then f(17)
                     else (f(17) ; run(m+1))
end;
run(0);
val ct=Time.toReal(Timer.checkRealTimer t);
```

Program 1 does not actually raise an exception at all because only small values, named 17, and so, are ever multiplied. Program 1 is for all practical purposes the same as program 2. The only difference between the two programs is that program 1 defines an exception handler and program 2 does not. We ran both programs modifying the test in the "if-statement" (n > 1000000), starting with 1000, then 50000, 100000, 200000, 400000, 600000, 800000 and finally 1000000. This controls the number of times the function **f** is called. The graph in figure 5.1 shows the time spent in program 1 (with exception handler) and program 2 (no exception handler). The difference between the curves in the graph shows the overhead when a program has an exception handler (even when no exception is raised).

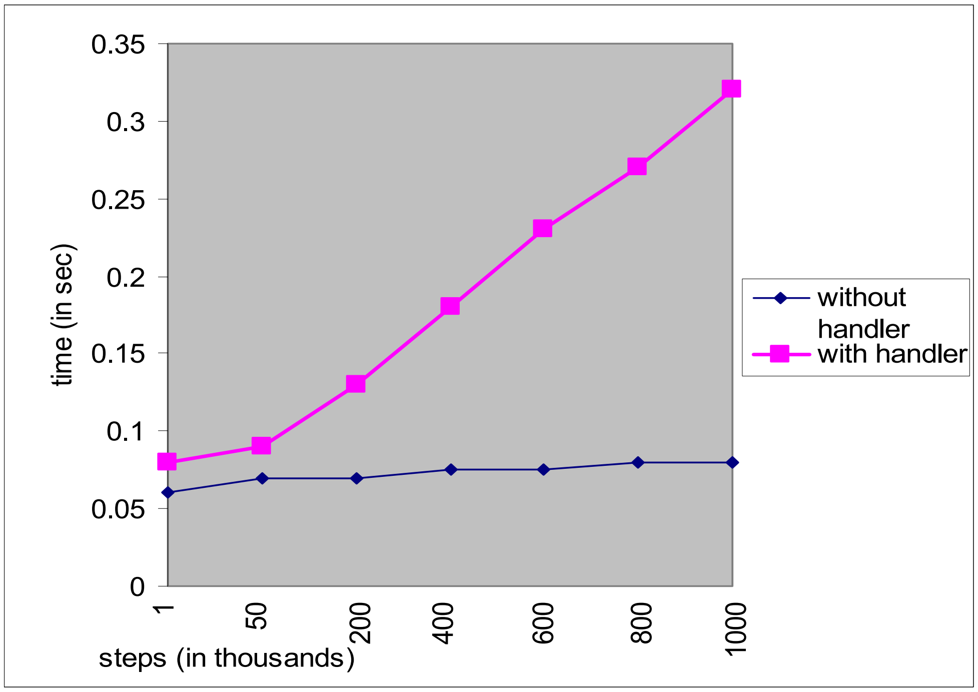

Figure 5.1 Comparison between using and not using exceptions in SML/NJ.

Looking at the code produced by the compiler for both programs (with and without handlers), we have 159 SPARC instructions (bcc, ld, st, jmp, etc) for the program with exception handling as opposed to 98 SPARC instructions for the program without exception handling.

5.4 Overhead in Exception Handling of OCAML

Implementing exception handling in OCAML also produces overhead. We tried the same experiment. We wrote essentially the same two programs in OCAML. The main difference between the implementation is the mechanism for getting the system time.

Program 1: Exception Handler

```
let
        f n = try n*n with overflow -> n
in
        let rec run m = if m>1000000 then f(17)
                        else (f(17);run(m+1))
        in
        run(0);;
let x1=times();;
print_float(x1.tms_utime);;print_newline();;
print_float(x1.tms_stime);;print_newline();;
```

Program 2: No Exception Handler

```
let
        f n =  n*n
in
        let rec run m = if m>1000000 then f(17)
                        else (f(17);run(m+1))
        in
        run(0);;
let x1=times();;
print_float(x1.tms_utime);;print_newline();;
print_float(x1.tms_stime);;print_newline();;
```

Again, as in SML, the only difference between these two programs is the exception handler. We ran both programs modifying the test. The graph in figure

5.2 shows the time spent in program 1 and program 2. The curves we got on the graph prove that the OCAML compiler also produces overhead when a program has exception handlers.

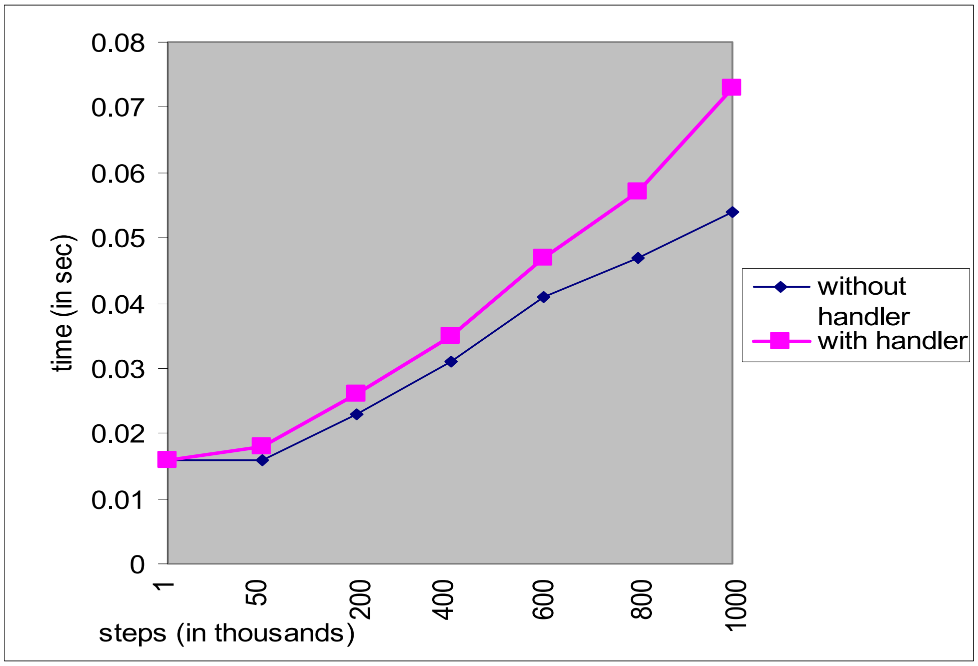

Figure 5.2 Comparison between using and not using exceptions in OCAML

We examined the assembly code of both programs. Program 1 has 78 SPARC instructions while program 2 has 41 instructions. Conclusions are difficult to draw, but CAML programs with exception handlers always ran slower than programs without handlers even when no exceptions were thrown.

5.5 The source of the Exception Handling Overhead in SML

We will explain the source of the overhead in SML programs, by presenting different stages in compilations of program 1 and program 2. The lambda code of program 1 is:

```
FIX
   (["f","run"],
    [FN
      ("n",
       HANDLE
         (APP (MULT,RECORD [VAR "n",VAR "n"]),
          FN
            ("e",
             COND
               (APP (EQ,RECORD [VAR "e",VAR "ovfl"]),VAR "n",
                RAISE (VAR "e"))))),
     FN
      ("x",
       COND
         (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
          APP
            (VAR "run",
             APP
               (PLUS,
                RECORD
                  [VAR "x",
                   APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])]))))],
    APP (VAR "run",INT 0))
```

Next, the compiler translates the source code (lambda) into a CPS program. This CPS program is later transformed into a flat program, which is just one large FIX definition of mutually recursive functions with no free variables, and no internal FIX operators. Here is the flattened CPS code for program 1.

```
FIX
  ([("f",["n","w1","x"],
     PRIMOP
      (gethdlr,[],["h2"],
       [PRIMOP
         (sethdlr,[VAR "n11"],[],
          [PRIMOP
            (*,[VAR "n",VAR "n"],["w12"],
             [PRIMOP
               (sethdlr,[VAR "h2"],[],
                [APP (VAR "k4",[VAR "w12",VAR "w1",VAR "x"])])])])])),
    ("k4",["x13","w1","x"],APP (VAR "w1",[VAR "x13",VAR "x"])),
    ("n11",["e3"],
     PRIMOP (sethdlr,[VAR "h2"],[],
      [APP (VAR "F5",[VAR "e3",VAR "k4"])])),
    ("F5",["e","k6"],
     PRIMOP(equal,[VAR "e",VAR "ovfl"],[],
       [APP (VAR "F7",[INT 0]),APP (VAR "F7",[INT 1])])),
    ("F7",["z8"],
     PRIMOP(equal,[VAR "z8",INT 0],[],
       [APP (VAR "k6",[VAR "n"]),
        PRIMOP (gethdlr,[],["h9"],[APP (VAR "h9",[VAR "e"])])])),
    ("run",["x","w1"],
     PRIMOP(equal,[VAR "x",INT 10],[],
       [APP (VAR "F14",[INT 0,VAR "x"]),
        APP (VAR "F14",[INT 1,VAR "x"])])),
    ("F14",["z15","x"],
     PRIMOP(equal,[VAR "z15",INT 0],[],
       [APP (VAR "f",[INT 17,VAR "r16",VAR "x"]),
        APP (VAR "f",[INT 17,VAR "r22",VAR "x"])])),
    ("r16",["x17","x"],APP (VAR "r25",[VAR "x17"])),
    ("r18",["x19"],APP (VAR "w1",[VAR "x19"])),
    ("r22",["x23","x"],
     PRIMOP
      (-,[VAR "x23",INT 288],["w21"],
       [PRIMOP
         (+,[VAR "x",VAR "w21"],["w20"],
          [APP (VAR "run",[VAR "w20",VAR "r18"])])])),
    ("r25",["x26"],APP (VAR "initialNormalCont",[VAR "x26"]))],
   APP (VAR "run",[INT 0,VAR "r25"]))
```

Function **f** starts executing two operations: a **gethdlr** and a **sethdlr**. The gethdlr operation stores the current handler in variable **h2**. The sethdlr operation installs the new handler **n11** (push). This handler is used only when an exception is raised (for example, an exception is thrown from the multiplication operation). After executing the multiplication operation, the old handler **h2** is reinstalled as the current handler (pop), and then a jump to function **k4** is performed.

The exception handling overhead is produced because these two operations **gethdlr** and **sethdlr** are always executed. They install a new handler even when an exception is never thrown in the program.

We found that another source of overhead is an extra function produced by the compiler named **k** (k4 in our CPS program). This function is used to invoke the continuation which is received by the function which declares the exception handler. But this code is invoked only if the exception is raised.

Now, let us see the lambda and CPS code of program # 2.

First, here is the lambda code:

```
FIX
  (["f","run"],
   [FN ("n",APP (MULT,RECORD [VAR "n",VAR "n"])),
    FN
      ("x",
       COND
         (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
          APP
            (VAR "run",
             APP
               (PLUS,
                RECORD
                [VAR "x",
                 APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])])))))],
    APP (VAR "run",INT 0))
```

Here is the flattened CPS code:

```
FIX
   ([("f",["n","w1","x"],
      PRIMOP
        (*,[VAR "n",VAR "n"],["w2"],
         [APP (VAR "w1",[VAR "w2",VAR "x"])])),
     ("run",["x","w1"],
      PRIMOP
        (equal,[VAR "x",INT 10],[],
         [APP (VAR "F3",[INT 0,VAR "x"]),
          APP (VAR "F3",[INT 1,VAR "x"])])),
     ("F3",["z4","x"],
      PRIMOP
        (equal,[VAR "z4",INT 0],[],
         [APP (VAR "f",[INT 17,VAR "r5",VAR "x"]),
          APP (VAR "f",[INT 17,VAR "r11",VAR "x"])])),
     ("r5",["x6","x"],APP (VAR "r14",[VAR "x6"])),
     ("r7",["x8"],APP (VAR "w1",[VAR "x8"])),
     ("r11",["x12","x"],
      PRIMOP
        (-,[VAR "x12",INT 288],["w10"],
         [PRIMOP
            (+,[VAR "x",VAR "w10"],["w9"],
             [APP (VAR "run",[VAR "w9",VAR "r7"])])])),
     ("r14",["x15"],APP (VAR "initialNormalCont",[VAR "x15"]))],
    APP (VAR "run",[INT 0,VAR "r14"]))
```

Because program 2 has no **handle** declaration, the CPS code has no instances where gethdlr and sethdlr are used. We can observe that function **f** has just one primitive operator which is the multiplication operator. This means program 2 executes faster than program 1.

In conclusion, exception handling in a program produces overhead because the operations, **gethdlr** and **sethdlr**, are executed even if an exception is never throw in the program. Our investigation found another possible source of overhead in function **k** (k4 in the CPS program), produced by the handle operation. This function is only used to invoke the continuation which is received by the function which declares the exception handler. We can observe in the CPS code of program 2 that this function does not exist. However, we believe that optimization of the compiler can avoid calling this function.

6 Zero Overhead Exception Handling

In chapter 5 we showed that some implementations incur overhead for using exception handling even when no exception are thrown. We also showed the results of some experiments with the SML/NJ and OCAML compilers, and then we identified and explained the source of the overhead in SML programs.

In this chapter we present the solution for the exception handling overhead problem. First, we explain a technique that some imperative language compilers use to implement exception handling. Particularly, we explain exception handling implementations using an exception table in Java and Ada. This technique uses a table where it keeps every possible exception to be handled along with related information for that exception. It is important to say that this technique has proven to deliver zero overhead exception handling.

Next, we present the implementation of the exception table technique in a functional programming language. This implementation presented some problems and/or weakness concerning dynamic propagation which are discussed and commented at the end of the section.

Last, we present a different approach that uses two continuations instead of one during the passing of parameters in the calling process. In this new approach one continuation encapsulates the rest of the normal continuation, and a second continuation is used for passing the abnormal computation. The second continuation is not passed as an extra argument but is passed as a displacement from the first continuation. This new technique can deliver zero overhead exception handling.

6.1 Exception table technique

Some imperative languages like Ada, C++, and Java have a different approach to implement exception handling [Din00, BR86, LYKPMEA, and Ven99]. We now describe how Java and Ada implement exception handling.

Java implementation of exception handling

We consider a program for computing the remainder.

```
static int remainder (int dividend, int divisor)
        throws OverflowException, DivideByZeroException  {
        if ((dividend = = Integer.MIN_VALUE) && (divisor = = -1))  {
                throw new OverflowException ( );
                try   {
                        return dividend % divisor;
                } catch (ArithmeticException e)  {
                        throw new DivideByZeroException ( );
                }
        }
}
```

The Java compiler generates the following bytecode sequence for the remainder method:

```
              0 iload_0
              1 ldc #1 <Integer –2147483648>
              3 if_icmpne 19
              6 iload_1
Body          7 iconst_m1
              8 if_icmpne 19
              11 new #4 <Class OverflowException>
              14 dup
              15 invokespecial #10 <Method OverflowException ( )>
              18 athrow
              19 iload_0
              20 iload_1
              21 irem
              22 ireturn
              23 pop
 Handler      24 new #2 <Class DivideByZeroException>
              27 dup
              28 invokespecial #9 <Method DivideByZeroException ( )>
              31 athrow
```

The bytecode sequence of the **remainder** method has two separate parts. The first part is the normal path of execution for the method. This part goes from pc offset zero through 22. The second part is the **catch** clause, which goes from pc offset 23 through 31. There appears to be no jump or entry into this part of the code; but as we will see, the runtime system may jump to this **catch** clause. It corresponds to the exception handler in the source program.

The **irem** instruction in the main bytecode sequence might throw an **ArithmeticException**. If this situation occurs, the Java virtual machine knows to jump to the bytecode sequence that implements the **catch** clause by looking up and finding the exception in a table. Each method that catches exceptions is associated with an exception table that is found in the class file along with the bytecode sequence of the method. The exception table has one entry for each exception that is caught by each **try** block. Each entry has four pieces of information:

- The start point
- The end point
- The pc offset within the bytecode sequence to jump to
- A constant pool index of the exception class that is being caught

The following is the exception table for the remainder method:

Exception table:

From	**to**	**target**	**type**
19	23	23	<Classjava.lang.ArithmeticException>

The preceding exception table indicates that from pc offset 19 through 22, inclusive, **ArithmeticException** is caught. The **try** block's endpoint value, listed in the table under the label "to," is always one more then the last pc offset for which the exception is caught. In this case the endpoint value is listed as 23, but the last pc offset for which the exception is caught is 22. This range, 19 to 22 inclusive, corresponds to the bytecode sequence that implements the code inside the **try** block of remainder function. The target listed in the preceding table is the pc offset to jump to if an **ArithmeticException** is thrown between the pc offsets 19 to 22, inclusive.

If an exception is thrown during the execution of a method, the Java virtual machine searches through the exception table for a matching entry. An exception table entry matches if the current program counter is within the range specified by the entry, and if the exception class thrown is the exception class specified by the entry (or is a subclass of the specified exception class). The Java virtual machine searches through the exception table in the order in which the entries appear in the table. When the first match is found, the virtual machine sets the program counter to the new pc offset location and continues execution there. If no match is found, the virtual machine pops the current stack frame and rethrows the same exception.

If no exception is thrown, the Java virtual machine continues the normal execution of the program, with no use of the exception table. Thus, the exception handler of the method (catch block) has no effect on the performance of the method.

Ada implementation of exception handling

In Ada [Ada95], an exception handler consists of a sequence of statements. Exception handlers appear in a case like structure at the end of a frame or block. For example,

```
Exception

        when E1 | E2 ➔ … ;            -- handler for E1 and E2

        when E3 ➔ … ;                 -- handler for E3

        when others ➔ … ;             -- handler for other exceptions
```

specifies a set of exceptions to which each handler applies.

The information in this case structure can be translated into code to be executed whenever an exception is raised or into a table to be searched by a recovery routine at runtime.

Some Ada compilers, like the DEC Ada compiler for the VAX/VMS system, implement responding to an exception by using a technique named **dynamic tracking** [BR86]. Here, the current context for exception handling is posted in a predictable location. Keeping this information up to date means that changes in the context for exception handling must be tracked dynamically. This technique needs a stack for storing the exception contexts. Upon entry to a frame, a new context record must be added to the top of the stack. Upon exit from a frame, the top context record must be popped from the stack. This technique is simple but produces overhead to normal execution. An alternative of dynamic tracking is to use a **static "map"** of the portion of memory that contains executable code, indicating the boundaries of each frame, and the boundaries of the sequence of statements within each frame. The map is implemented as a **table of exceptions**. Constructing the table requires knowledge of the exact address of each contiguous block of code for each frame. It must therefore involve cooperation of the compiler, the linker, and the virtual address translator. In particular, any relocation of code modules must be reflected by corresponding adjustments to the map. This method does not produce any overhead on normal execution, but requires a degree of coordination of compilation, linking, loading, and virtual address translation. The Intermetric Ada compiler for the IBM mainframes has a particular simple scheme for a mix of static and dynamic tracking.

Implementation of exception handling using tables in ML

The main idea of this implementation is to use an exception table where we can keep in each entry of the table, the range of the protected code, the address of the exception handler, and the exception name. Now, the implementation in ML would be different because the structure of a functional language is different than the structure of an imperative language. Instead of using blocks like "try and catch," ML uses expressions. The syntax for an expression of type "handler" in SML:

<expression> **handle** <match>

where <expression> can be an *id*, or a *const* expression, an *if* expression, etc.; and <match> can be a pattern, followed by the symbol "=>" and an expression. As we mentioned before, CAML uses a notation for defining exception handlers similar to SML.

The exception table is created when abstract machine code is generated. We describe a machine-level translation of a program with exception handling to produce code along with the exception table:

- Eliminate the primitive operators which cause the overflow in the program (**gethdlr** and **sethdlr** operators).
- Obtain the range of the protected code
- Produce machine code for the protected code
- For user-defined exceptions insert a Jump instruction (its address will be the first entry of the corresponding exception table). For system-defined exceptions the compiler will not produce a Jump instruction. This kind of

exception is raised by the run-time system, even though a handler in the program can catch them.

- An entry in the exception table is inserted. The entry content is the range of the protected code, the address of the corresponding handle, and the exception name.
- Finally, we make a small optimization in order to avoid making two consecutive jumps; that is means calling function "k" ("k4" in the last program), from outside of the handler.

We show in the next table part of the final code (abstract machine code) produced by our compiler. If an exception is thrown during the execution of the expression n*n (line 6 of code), our runtime system searches through the exception table (line 74) for a matching entry. An exception table entry matches, when the current program counter is within the range specified by the entry (in this example the PC is pointing to line 6), and the exception type thrown is one of the exceptions specified by the entry ("ovfl" was specified in the SML code). The simulator searches through the exception table in the order in which entries appear in the table (just one entry for this example). When the first match is found, the run-time system sets the program counter to the new pc offset location (it takes the address of the handler "n10" which is 16), and continues the execution.

```
(* MACHINE CODE WITH EXCEPTION *)

0               LOAD    Const 0,Reg 1
1               LOAD    Mem r24,Reg 2
2               JUMP    run
        LAB f:
3               STORE   Reg 1,Mem w1
4               STORE   Reg 2,Mem n
5               STORE   Reg 3,Mem x
6               MUL     Mem n,Mem n,
                        Mem w11
7               LOAD    Mem w11,Reg 1
8               LOAD    Mem x,Reg 2
9               JUMP    w1
        LAB k4:
10              STORE   Reg 1,Mem x12
11              STORE   Reg 2,Mem w1
12              STORE   Reg 3,Mem x
13              LOAD    Mem x12,Reg 1
14              LOAD    Mem x,Reg 2
15              JUMP    w1
        LAB n10:
16              STORE   Reg 1,Mem e3
17              LOAD    String h2,
                        Reg 99
18              LOAD    Mem e3,Reg 1
19              LOAD    Mem k4,Reg 2
20              JUMP    F5
                .
                .
        LAB run:
37              STORE   Reg 1,Mem x
38              STORE   Reg 2,Mem w1
39              CJUMP   EQ,Mem x,Const
                        10,L4,L5
        LAB L4:
40              LOAD    Const 0,Reg 1
41              LOAD    Mem x,Reg 2
42              JUMP    F13
                .
                .
        LAB r15:
57              STORE   Reg 1,Mem x16
58              STORE   Reg 2,Mem x
59              LOAD    Mem x16,Reg 1
60              JUMP    r24
        LAB r17:
61              STORE   Reg 1,Mem x18
62              LOAD    Mem x18,Reg 1
63              JUMP    w1
        LAB r21:
64              STORE   Reg 1,Mem x22
65              STORE   Reg 2,Mem x
66              SUB     Mem x22,Const
                        288,Mem w20
67              ADD     Mem x,Mem w20,
                        Mem w19
68              LOAD    Mem w19,Reg 1
69              LOAD    Mem r17,Reg 2
70              JUMP    run
        LAB r24:
71              STORE   Reg 1,Mem x25
72              LOAD    Mem x25,Reg 1
```

```
(* MACHINE CODE WITH NO EXCEPTION *)

0               LOAD    Const 0,Reg 1
1               LOAD    Mem r14,Reg 2
2               JUMP    run
        LAB f:
3               STORE   Reg 1,Mem n
4               STORE   Reg 2,Mem w1
5               STORE   Reg 3,Mem x
6               MUL     Mem n,Mem n,
                Mem w2
7               LOAD    Mem w2,Reg 1
8               LOAD    Mem x,Reg 2
9               JUMP    w1
        LAB run:
10              STORE   Reg 1,Mem x
11              STORE   Reg 2,Mem w1
12              CJUMP   EQ,Mem x,
                        Const 10,L0,L1
        LAB L0:
13              LOAD    Const 0,Reg 1
14              LOAD    Mem x,Reg 2
15              JUMP    F3
        LAB L1:
16              LOAD    Const 1,Reg 1
17              LOAD    Mem x,Reg 2
18              JUMP    F3
        LAB F3:
19              STORE   Reg 1,Mem z4
20              STORE   Reg 2,Mem x
21              CJUMP   EQ,Mem z4,
                        Const 0,L2,L3
        LAB L2:
22              LOAD    Const 17,Reg 1
23              LOAD    Mem r5,Reg 2
24              LOAD    Mem x,Reg 3
25              JUMP    f
        LAB L3:
26              LOAD    Const 17,Reg 1
27              LOAD    Mem r11,Reg 2
28              LOAD    Mem x,Reg 3
29              JUMP    f
        LAB r5:
30              STORE   Reg 1,Mem x6
31              STORE   Reg 2,Mem x
32              LOAD    Mem x6,Reg 1
33              JUMP    k
        LAB r7:
34              STORE   Reg 1,Mem x8
35              LOAD    Mem x8,Reg 1
36              JUMP    w1
        LAB r11:
37              STORE   Reg 1,Mem x12
38              STORE   Reg 2,Mem x
39              SUB     Mem x12,
                        Const 288,
                        Mem w10
40              ADD     Mem x,Mem w10,
                        Mem w9
41              LOAD    Mem w9,Reg 1
42              LOAD    Mem r7,Reg 2
43              JUMP    run
        LAB r14:
```

`73          JUMP    k`	`44          STORE   Reg 1,Mem x15`
`       LAB ETab:`	`45          LOAD    Mem x15,Reg 1`
`74          ExT     3,6,n10,ovfl`	`46          JUMP    k`
`       LAB end:`	`       LAB end:`

If no match is found, the run-time system rethrows the same exception from the point where the last call was executed. Now, if no exception is thrown during the execution of the multiplication expression, the program jump to k4 (line 10), which is a continuation from the **handle** operation.

Implementation weakness

We tested our implementation with different examples, finding major difficulties concerning dynamic propagation. As we explained in chapter two, an exception not handled is automatically raised to other frames along the calling chain until a handler is found or until a program boundary is reached. This propagation method is called **automatic or dynamic propagation**. If the handler for a raised exception cannot be found locally, a Java, Ada or C++ program unwinds the stack of the try block and propagates the exception to its caller. This procedure continues until a handler is found or until a default handler is called, which then aborts the program.

Our implementation creates the exception table during the compilation of the program. With the help of the table the program can find the handler if it is in the same context. However, when the handler for the thrown exception is in another context or frame from the calling chain then the table does not work. In order to implement dynamic propagation, we use a different approach: using two continuation for normal and abnormal (exception) procedures. In the next section we explain that approach.

6.2 Low overhead using two continuations

Now, we present an implementation which can be used with programs where dynamic propagation occurs. This exception handling implementation can deliver a good performance in handling exceptions where we obtain a decrement of the overhead with respect to the normal method of implementation (see chapter 5).

In the analysis we made in chapter 5, we concluded that unnecessary utilization of gethdlr and sethdlr operators (along with the use of function **k**), produces the main overhead of SML programs. Those two operators are used to save and restore the handlers of the programs. We designed a different CPS conversion program, where those operators (gethdlr and sethdlr) are not longer needed. The main idea of the new CPS code is the use of a second continuation for abnormal situations. The next example will serve to illustrate the new technique. We will show again the SML, lambda, and CPS code.

The SML code of the program is

```
exception mult
let
        fun f(n)= h(n) handle mult => n
        fun h(m)= g(m)
        fun g(v)= (raise mult) handle ovfl => v
in
        f(17)
end
```

and the respective lambda code is

```
APP
  (FN
    ("mult",
     FIX
      (["f","h","g"],
       [FN
         ("n",
          HANDLE
           (APP (VAR "h",VAR "n"),
            FN
             ("e1",
              COND
               (APP (EQ,RECORD [VAR "e1",VAR "mult"]),VAR "n",
                RAISE (VAR "e1"))))),
         FN ("m",APP (VAR "g",VAR "m")),
        FN
         ("v",
          HANDLE
           (RAISE (VAR "mult"),
            FN
             ("e2",
              COND
               (APP (EQ,RECORD [VAR "e2",VAR "ovfl"]),VAR "v",
                RAISE (VAR "e2")))))],APP (VAR "f",INT 17))),
   APP (MAKEREF,STRING "MUL"))
```

Observing the lambda code, we see in boldface the **handler** defined in the SML function **f**. A handler with the old technique produces the operations **gethdlr** and **sethdlr** which cause the main overhead in a program. With the new approach or design of the conversion program the handler is passed as a second continuation (we call it abnormal continuation). This second continuation could be used if a raise operation of such exception is executed.

The CPS code with the new technique is

```
FIX
  ([("r1",["x2"],APP (VAR "initialNormalCont",[VAR "x2"])),

   ("e'3",["z4"],APP (VAR "initialAbnormalCont",[VAR "z4"])),

   ("F5",["mult","k6","ak7"],
       APP (VAR "f",[a.INT 17,VAR "r26",VAR "e'28",VAR "mult"])),

   ("f",["n","w8","ak9","mult"],
       APP (VAR "h",[VAR "n",VAR "r14",VAR "e'16",VAR "mult"])),

   ("H10",["e1","mult","v"],
       PRIMOP(equal,[VAR "e1",VAR "mult"],[],
         [APP (VAR "F11",[a.INT 0,VAR "v"]),
          APP (VAR "F11",[a.INT 1,VAR "v"])])),

   ("F11",["z12","n"],
       PRIMOP(equal,[VAR "z12",a.INT 0],[],
         [APP (VAR "r26",[VAR "n"]),
          APP (VAR "ak9",[VAR "e1"])])),

   ("r14",["x15"],APP (VAR "w8",[VAR "x15"])),

   ("e'16",["z17","mult","v"],APP (VAR "H10",[VAR "z17",VAR "mult",VAR "v"])),

   ("h",["m","w8","ak9","mult"],
         APP (VAR "g",[VAR "m",VAR "r18",VAR "e'20",VAR   "mult"])),

   ("r18",["x19"],APP (VAR "w8",[VAR "x19"])),

   ("e'20",["z21","mult","v"],APP (VAR "e'16",[VAR "z21",VAR "mult",VAR "v"])),


   ("g",["v","w8","ak9","mult"],
         APP (VAR "H22",[VAR "mult",VAR "w8",VAR "v",VAR "mult"])),

   ("H22",["e2","w8","v","mult"],
       PRIMOP(equal,[VAR "e2",VAR "ovfl"],[],
         [APP (VAR "F23",[a.INT 0,VAR "w8",VAR "v",VAR "e2",VAR "mult"]),
          APP (VAR "F23",[a.INT 1,VAR "w8",VAR "v",VAR "e2",VAR "mult"])])),


   ("F23",["z24","w8","v","e2","mult"],
       PRIMOP(equal,[VAR "z24",a.INT 0],[],
         [APP (VAR "w8",[VAR "v",VAR "mult"]),
          APP (VAR "e'20",[VAR "e2",VAR "mult",VAR "v"])])),

   ("r26",["x27"],APP (VAR "r1",[VAR "x27"])),

   ("e'28",["z29"],APP (VAR "ak7",[VAR "z29"]))],

   PRIMOP(makeref,[a.STRING "MUL"],["w30"],
       [APP (VAR "F5",[VAR "w30",VAR "r1",VAR "e'3"])]))
```

When function **h** is invoked (first boldface text), the normal continuation (**r14**) and the abnormal continuations (**e'16**) are passed as arguments. This second continuation, is the name of a function whose main body contains an invocation to the handler **H10** and will be used later in the program. Function **h**, then just invokes function **g** passing the two continuations **r18** and **e'20**. Last, function g has an invocation to the handler **H22** (produced by the **RAISE** operation of the lambda program). This handler, **H22**, will not catch the exception, so an invocation to the first handler (**H10**) will be performed by using the abnormal continuation contained in variable **ak9**.

The new conversion program

In chapter 3, we presented the algorithm to convert a lambda expression into a corresponding continuation expression. The conversion function **f** takes two arguments: a lambda expression and a continuation function. The result is a CPS expression [App92]. Our new approach is a modification to the mutually-recursive function ***f*** of Appel. We modified the translation by adding a new continuation for abnormal computations. The **new** function ***convert*** takes three arguments: a lambda expression, a continuation function **c** for normal computations, and a continuation function **e** for abnormal computations.

All functions in the lambda language have exactly one argument. When translated to CPS with one continuation, the continuation adds one argument. Now in the translation with two continuation every function become a function with three arguments.

The first part of the algorithm is the same than in the old algorithm (like function **f** of Appel [App92] we need to create new variables).

Function **f** (we call it now **convert**) takes two continuations functions (**c** and **e**) as arguments, besides the lambda expression.

```
fun convert(lambda.VAR v, c, e) = c(VAR v)
|   convert(lambda.INT i, c, e) = c(INT i)
|   convert(lambda.STRING s,c, e) = c(STRING s)
```

A lambda HANDLE definition translates into a single CPS function definition. This function is given a name **H**, and corresponds to the handler of the produced continuation expression. The body of **H** is given by converting subexpression **B** (the second operand of the HANDLE expression) into a CPS expression. Last, the main body of the FIX expression is the result of translating the first operand **A** of the HANDLE expression. In the translation of **A**, the continuation function **e'** is used as a third argument. If **A** contains a RAISE expression then this function **e'** is used as the normal continuation. Then, when

A is done executing, it will bind its result to some variable **w** and then apply **H** to that variable.

```
convert(HANDLE (A,FN(z,B)),c, e) =
let
        val H = newVar ("H")
        fun e' w = APP(VAR H,[w])
in
        FIX([(H,[z],convert(B,c , e))], convert (A,c,e'))
end
```

A RAISE expression is converted by applying the abnormal continuation into the expression **E**. Because the abnormal continuation is a handler, then the result of the translations is the invocation of that handler.

```
convert(RAISE E, c, e) = convert(E,e,e)
```

In the case of function definitions, the modifications (shown in boldface below) consist in adding a third argument **ak** for the abnormal continuation.

```
convert(FN (v,E), c, e) =
let
   val F  = newVar ("F");
   val k  = newVar ("k");
   val ak = newVar ("ak");
   fun e' z = APP(VAR ak,[z])
in
   FIX([(F,[v,k,ak],convert (E,(fn z=>APP
      (VAR k,[z])), e'))],c(VAR F))
end
```

With respect to lambda expressions constructed using **FIX**, the modifications again consist of adding a third argument to each mutually recursive function.

```
convert(FIX(hx,bx,E),c, e) =
let
   val w = newVar ("w")
   val ak = newVar ("ak")
   fun e" w = APP(VAR ak,[w])
   fun g(h1::h,FN(v,B)::b)=
       (h1,[v,w,ak], convert (B, fn z=>
                  APP(VAR w,[z]), e"))::g(h,b)
   |   g(nil,nil) = nil
in
   FIX (g(hx,bx),convert (E,c, e))
end
```

In a function call, we need to add a new return address **e'** for abnormal continuation. This address will be used when an exception is raised and the abnormal continuation is taken as the normal continuation to execute.

```
convert(APP (F,E), c, e) =
let
   val r= newVar ("r");
   val x= newVar ("x");
   val e'= newVar ("e'");
   val z= newVar ("z");
in
  FIX([(r,[x],c(VAR x)),
          (e',[z],e(VAR z))],
           convert(F,(fn f2=>convert(E,(fn e2=>APP(f2, [e2,VAR r,VAR e'])), e)), e))
end
```

In the rest of the cases, the only modification to make is adding the third continuation in the list of calling and receiving arguments .

6.3 Zero overhead with one continuation and displacement

The use of an extra argument for the abnormal continuation, provides better performance in normal execution of programs with exception declarations. However, the extra argument imposes some overhead of its own during the normal execution of a program. We can remove that overhead by making some small changes to the compiler; specifically the program that converts to CPS code. The main idea is passing only one continuation address for the normal and abnormal continuations. Both continuation addresses are consecutive functions whose names differ only by an apostrophe. Next, we illustrate this technique by using the same example shown in the previous section, but with the modifications discussed before.

```
FIX
  ([("r1",["x2"],APP (VAR "initialNormalCont",[VAR "x2"])),

    ("r1'",["z3"],APP (VAR "initialAbnormalCont",[VAR "z3"])),

    ("F4",["mult","k5"],
      APP (VAR "f",[a.INT 17,VAR "r23",VAR "mult"])),

    ("f",["n","w7","mult"],
      APP (VAR "h",[VAR "n",VAR "r13",VAR "mult"])),

    ("H9",["e1","mult","v"],
      PRIMOP(equal,[VAR "e1",VAR "mult"],[],
        [APP (VAR "F10",[a.INT 0,VAR "v"]),
         APP (VAR "F10",[a.INT 1,VAR "v"])])),

    ("F10",["z11","n"],
      PRIMOP(equal,[VAR "z11",a.INT 0],[],
        [APP (VAR "r23",[VAR "n"]),
         APP (VAR "w7",[VAR "e1"])])),
    ("r13",["x14"],APP (VAR "w7",[VAR "x14"])),

    ("r13'",["z15","mult","v"],APP (VAR "H9",[VAR "z15",VAR "mult",VAR "v"])),

    ("h",["m","w7","mult"],APP (VAR "g",[VAR "m",VAR "r16",VAR "mult"])),

    ("r16",["x17"],APP (VAR "w7",[VAR "x17"])),

    ("r16'",["z18","mult","v"],APP (VAR "r13'",[VAR "z18",VAR "mult",VAR "v"])),

    ("g",["v","w7","mult"],APP (VAR "H19",[VAR "mult",VAR "w7",VAR "v",VAR "mult"])),

    ("H19",["e2","w7","v","mult"],
        PRIMOP(equal,[VAR "e2",VAR "ovfl"],[],
         [APP (VAR "F20",[a.INT 0,VAR "w7",VAR "v",VAR "e2",VAR "mult"]),
          APP (VAR "F20",[a.INT 1,VAR "w7",VAR "v",VAR "e2",VAR "mult"])])),

    ("F20",["z21","w7","v","e2","mult"],
        PRIMOP(equal,[VAR "z21",a.INT 0],[],
         [APP (VAR "w7",[VAR "v",VAR "mult"]),
          APP (VAR "r16'",[VAR "e2",VAR "mult",VAR "v"])])),
    ("r23",["x24"],APP (VAR "r1",[VAR "x24"])),

    ("r23'",["z25"],APP (VAR "ak6",[VAR "z25"]))],

    PRIMOP(makeref,[a.STRING "MUL"],["w26"],
        [APP (VAR "F4",[VAR "w26",VAR "r1"])]))
```

The program starts invoking function **F4**, and passing **w26** and **r1**, the return address from normal continuation. There is no need to pass **r1'**, the return address from abnormal continuation, because it can be referenced by just adding the apostrophe to the name of the normal continuation. Next, from function **F4** there is an invocation to function **f** where the normal continuation **r23** is passed as the second argument. Then, there is an invocation to function h passing the continuation **r13** as second argument, and from function **h** there is a call to function **g** passing continuation **r16** as one of the arguments. Next, function **g** calls the function **H19** function passing continuation **w7** (which contains the value **r16**) as the second argument. In function **H19**, the condition checks if **e2** (exception mult) is equal to **ovfl** (overflow). Because it is false, then there is a jump to **F20** passing again continuation **w7**. The condition of function **F20** only checks the validity of last condition (function H19). Because it was false then it calls for function **r16'** (the abnormal continuation) instead of function **r16** (the normal continuation). Function **r16'** invokes function **r13'** (abnormal continuation too), and function **r13'** calls for function **H9**, that is the handler of function **f** (see last SML code). The condition of the handler function **H9** is true (the raised exception is equal to mult exception), and the program finishes calling functions F10, r23, and r1 consecutively.

Problems due to optimization

This method does not impose any obvious overhead on normal execution. On the other hand, a problem in a real compiler (like SML/NJ) is that during optimization the compiler relocates consecutive functions of the program. This problem is also faced by others compilers that use exception tables [BR86, Din00]. As a solution to this problem we mentioned before that Ada [BR86] compilers interact with the linker, the loader, and the virtual address translator. The Ada compiler uses a static "map" of the portion of memory that contains executable code, indicating the boundaries of each frame, and the boundaries of the sequence of statements within each frame. The map is implemented as an exception table. Constructing the table requires knowledge of the exact address of each contiguous block of code for each frame. Any relocation of the code during optimization must be reflected by corresponding adjustments to the table. A global static table corresponds to each main program. The table contains information like the low address of the segment (one segment is the code of a sequence of statements, an exception handler, etc.), the address of the code to be executed (handler) when an exception is raised within the code segment beginning with this lower address, etc. If the optimizer of the compiler relocates the code, then the linker/loader modifies the information in the table by adding or subtracting a constant to the low addresses of code segment and the exception handler address. The disadvantage of this technique is the dependency of the compiler with the linker and other programs. On the other hand, some C++ compilers like the HP C++ compiler follow a different method for dealing with the problem of relocation due to optimizations [Din00]. They also use exception-handling tables. They divide the code of a program in exception-handling code and normal code. The normal or nonexceptional code is optimized while compensation code is placed along the exceptional path to restore program state (before executing the exception handlers) to what it would have been if the optimization had not taken place. The place where such compensation code is added is called a **landing pad**, which serves as an alternate return path for each call.

In our compiler, we can work with this last approach; that is, including compensation code, in order to restore the program state to what it would be if optimization had not been done in the main control flow (see figure 6.1).

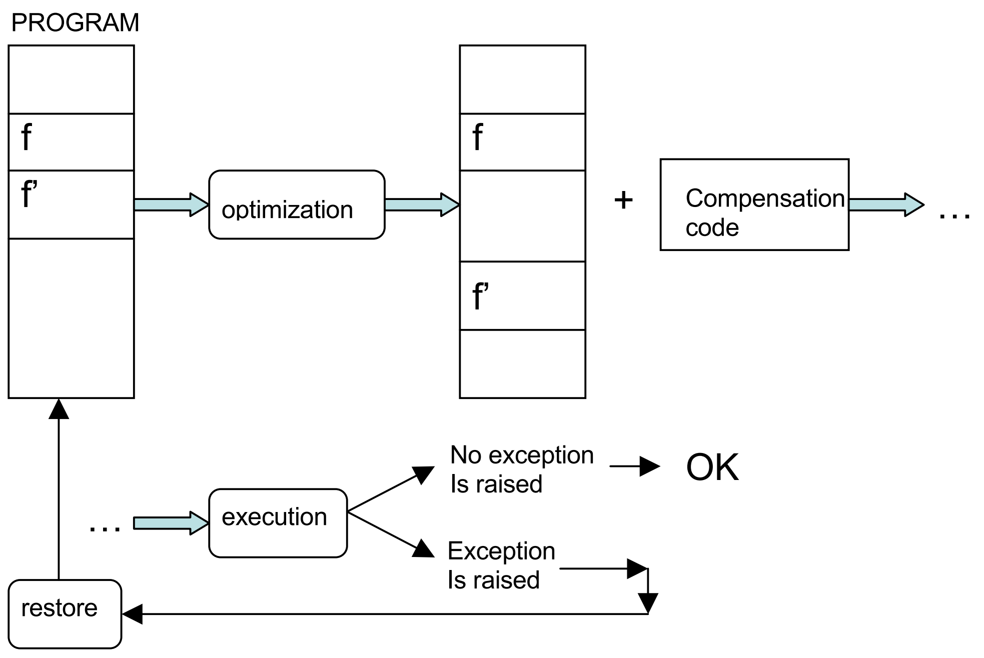

Figure 6.1 Inclusion of compensation code

If an exception is raised the compensation code would do some operations, the most important would be to restore the program state to what it would be if optimization had not been done in the main control flow. The implementation of this method has not yet been accomplished.

Another solution to the problem is including a block with two jump operations (**e** and **e'**) before the code of the functions for normal and abnormal continuations. One **jump** will be to the normal continuation **f**; another **jump** will be to the abnormal continuation **f'**. Then, every reference to the normal or abnormal continuations from the main program will be first a reference to that block of consecutive jumps **e** and **e'**.

7 Experiments and Performance

This chapter presents a set of examples or experiments we made in order to test the performance of the new approaches. We also present a set of graphs comparing the performance between the traditional technique of exception handling and the new techniques that we designed and implemented. The set of graphs demonstrate that a functional program using exception handling can reach low and zero overhead when using the two-continuations and one-continuation-displacement techniques respectively.

7.1 Experimental measurements

All measurement in the experiments are made by counting the number of instructions executed by the simulated programs. As we explained in chapter 4, we implemented a simulator of a real machine. The simulator "executes" a program in abstract machine code, and obtain information like the number of instructions performed by that program. The abstract machine code can be produced by three different compilers. One using the original approach (using the gethdlr and sethdlr operators), a second that uses the two-continuations approach, and a third that uses the one-continuation-displacement approach. Some comparisons in the experiments are between programs that declare and use exception handlers; and some are between programs that declare and never use exception handlers which is, of course, the most important situation for our research.

7.2 Experimental examples

Next, we present a set of examples to be used in the experiments. For each program we show the code in SML language (for clarity reasons), lambda code, and CPS (flat) code. The abstract machine code of that programs is found in the appendix.

The next program was compiled by using the original (old) approach in the translation of lambda expressions to CPS code. The program has three functions (f, h, and g), where two of them (f and g) have handler definitions. The program start by calling function f. This function in turn call function h, and then function h call function g. This last function raises an exception (exception **mult**). The handler of the last function (g) can not catch the exception **mult**. So, the exception is propagated by following in reverse order the actual calling chain, until the exception is caught by the handler of function f.

PROGRAM 1 - Old technique

SML code

```
exception mult
let
        fun f(n)= h(n) handle mult => n
        fun h(m)= g(m)
        fun g(v)= (raise mult) handle ovfl => v
in
        f(17)
end
```

Lambda code

```
APP
   (FN
     ("mult",
      FIX
        (["f","h","g"],
         [FN
           ("n",
            HANDLE
              (APP (VAR "h",VAR "n"),
               FN
                 ("e1",
                  COND
                    (APP (EQ,RECORD [VAR "e1",VAR "mult"]),VAR "n",
                     RAISE (VAR "e1"))))),FN ("m",APP (VAR "g",VAR "m")),
          FN
           ("v",
            HANDLE
              (RAISE (VAR "mult"),
               FN
                 ("e2",
                  COND
                    (APP (EQ,RECORD [VAR "e2",VAR "ovfl"]),VAR "v",
                     RAISE (VAR "e2")))))],APP (VAR "f",INT 17))),
   APP (MAKEREF,STRING "MUL"))
```

CPS code

```
 FIX
  ([("r1",["x2"],APP (VAR "initialNormalCont",[VAR "x2"])),

    ("F3",["mult","k4"],APP (VAR "f",[a.INT 17,VAR "r33",VAR "mult"])),

    ("f",["n","w5","mult"],
       PRIMOP(gethdlr,[],["h6"],
         [PRIMOP(sethdlr,[VAR "n15"],[],
            [APP (VAR "h",[VAR "n",VAR "r16",VAR "mult",VAR "h6"])])])),

    ("k8",["x18"],APP (VAR "r33",[VAR "x18"])),

    ("n15",["e7","h6","v","mult"],
       PRIMOP(sethdlr,[VAR "h6"],[],
         [APP (VAR "F9",[VAR "e7",VAR "k8",VAR "mult",VAR "v"])])),

    ("F9",["e1","k10","mult","n"],
       PRIMOP(equal,[VAR "e1",VAR "mult"],[],
         [APP (VAR "F11",[a.INT 0,VAR "n"]),
          APP (VAR "F11",[a.INT 1,VAR "n"])])),
    ("F11",["z12","n"],
       PRIMOP(equal,[VAR "z12",a.INT 0],[],
         [APP (VAR "k8",[VAR "n"]),
          PRIMOP(gethdlr,[],["h13"],
              [APP (VAR "h13",[VAR "e1"])])])),

    ("r16",["x17"],
       PRIMOP(sethdlr,[VAR "h6"],[],
         [APP (VAR "k8",[VAR "x17"])])),

    ("h",["m","w5","mult","h6"],
          APP (VAR "g",[VAR "m",VAR "r19",VAR "mult",VAR "h6"])),

    ("r19",["x20"],APP (VAR "w5",[VAR "x20"])),

    ("g",["v","w5","mult","h6"],
       PRIMOP(gethdlr,[],["h21"],
         [PRIMOP(sethdlr,[VAR "n30"],[],
             [PRIMOP(gethdlr,[],["h31"],
                [APP (VAR "h31",
                       [VAR "mult",VAR "h21",VAR "h6",VAR "v",VAR "mult"])])])])),

    ("k23",["x32"],APP (VAR "w5",[VAR "x32"])),

    ("n30",["e22","h21","h6","v","mult"],
       PRIMOP(sethdlr,[VAR "h21"],[],
         [APP (VAR "F24",[VAR "e22",VAR "k23",VAR "h6",VAR "v",VAR "mult"])])),

    ("F24",["e2","k25","h6","v","mult"],
       PRIMOP(equal,[VAR "e2",VAR "ovfl"],[],
         [APP (VAR "F26",[a.INT 0,VAR "e2",VAR "h6",VAR "v",VAR "mult"]),
          APP (VAR "F26",[a.INT 1,VAR "e2",VAR "h6",VAR "v",VAR "mult"])])),

    ("F26",["z27","e2","h6","v","mult"],
       PRIMOP(equal,[VAR "z27",a.INT 0],[],
         [APP (VAR "k25",[VAR "v"]),
          PRIMOP(gethdlr,[],["h28"],
                 [APP (VAR "h28",[VAR "e2",VAR "h6",VAR "v",VAR "mult"])])])),

    ("r33",["x34"],APP (VAR "r1",[VAR "x34"]))],

    PRIMOP(makeref,[a.STRING "MUL"],["w35"],
       [APP (VAR "F3",[VAR "w35",VAR "r1"])]))
```

Now, program 1 is compiled by using the new approach in the translation of lambda expressions to CPS code. In the new approach two continuations are produced when producing the CPS code. We can observe, in the next CPS code, two continuations. One is for passing normal computations (the rest of the function computation), and another one for passing abnormal computations (exception handlers).

PROGRAM 1- Two-continuation technique

SML code

```
exception mult
let
        fun f(n)= h(n) handle mult => n
        fun h(m)= g(m)
        fun g(v)= (raise mult) handle ovfl => v
in
        f(17)
end
```

Lambda code

```
APP
   (FN
      ("mult",
       FIX
         (["f","h","g"],
          [FN
             ("n",
              HANDLE
                (APP (VAR "h",VAR "n"),
                 FN
                   ("e1",
                    COND
                      (APP (EQ,RECORD [VAR "e1",VAR "mult"]),VAR "n",
                       RAISE (VAR "e1"))))),FN ("m",APP (VAR "g",VAR "m")),
       FN
         ("v",
          HANDLE
            (RAISE (VAR "mult"),
             FN
               ("e2",
                COND
                  (APP (EQ,RECORD [VAR "e2",VAR "ovfl"]),VAR "v",
                   RAISE (VAR "e2"))))))],APP (VAR "f",INT 17))),
   APP (MAKEREF,STRING "MUL"))
```

CPS code

```
FIX
   ([("r1",["x2"],APP (VAR "initialNormalCont",[VAR "x2"])),

    ("e'3",["z4"],APP (VAR "initialAbnormalCont",[VAR "z4"])),

    ("F5",["mult","k6","ak7"],
        APP (VAR "f",[a.INT 17,VAR "r26",VAR "e'28",VAR "mult"])),

    ("f",["n","w8","ak9","mult"],
        APP (VAR "h",[VAR "n",VAR "r14",VAR "e'16",VAR "mult"])),

    ("H10",["e1","mult","v"],
        PRIMOP(equal,[VAR "e1",VAR "mult"],[],
           [APP (VAR "F11",[a.INT 0,VAR "v"]),
            APP (VAR "F11",[a.INT 1,VAR "v"])])),

    ("F11",["z12","n"],
        PRIMOP(equal,[VAR "z12",a.INT 0],[],
           [APP (VAR "r26",[VAR "n"]),
            APP (VAR "ak9",[VAR "e1"])])),

    ("r14",["x15"],APP (VAR "w8",[VAR "x15"])),

    ("e'16",["z17","mult","v"],APP (VAR "H10",[VAR "z17",VAR "mult",VAR "v"])),

    ("h",["m","w8","ak9","mult"],
          APP (VAR "g",[VAR "m",VAR "r18",VAR "e'20",VAR "mult"])),

    ("r18",["x19"],APP (VAR "w8",[VAR "x19"])),

    ("e'20",["z21","mult","v"],APP (VAR "e'16",[VAR "z21",VAR "mult",VAR "v"])),

    ("g",["v","w8","ak9","mult"],
          APP (VAR "H22",[VAR "mult",VAR "w8",VAR "v",VAR "mult"])),

    ("H22",["e2","w8","v","mult"],
        PRIMOP(equal,[VAR "e2",VAR "ovfl"],[],
           [APP (VAR "F23",[a.INT 0,VAR "w8",VAR "v",VAR "e2",VAR "mult"]),
            APP (VAR "F23",[a.INT 1,VAR "w8",VAR "v",VAR "e2",VAR "mult"])])),

    ("F23",["z24","w8","v","e2","mult"],
        PRIMOP(equal,[VAR "z24",a.INT 0],[],
           [APP (VAR "w8",[VAR "v",VAR "mult"]),
            APP (VAR "e'20",[VAR "e2",VAR "mult",VAR "v"])])),

    ("r26",["x27"],APP (VAR "r1",[VAR "x27"])),

    ("e'28",["z29"],APP (VAR "ak7",[VAR "z29"]))],

    PRIMOP(makeref,[a.STRING "MUL"],["w30"],
        [APP (VAR "F5",[VAR "w30",VAR "r1",VAR "e'3"])]))
```

Last, program 1 is compiled by using the modification to the new approach in the translation of lambda expressions to CPS code. The new CPS code produced by this compilation changes the name of some functions (functions which computes abnormal continuations). The new name is similar to the name of the previous function that computes the normal continuation. Another important modification to the CPS code is that now it only passes as a parameter the normal continuation (the abnormal continuation can be referenced or located using a displacement from the normal continuation).

PROGRAM 1- One-continuation-displacement technique

SML code

```
exception mult
let
        fun f(n)= h(n) handle mult => n
        fun h(m)= g(m)
        fun g(v)= (raise mult) handle ovfl => v
in
        f(17)
end
```

Lambda code

```
APP
   (FN
     ("mult",
      FIX
        (["f","h","g"],
         [FN
            ("n",
             HANDLE
               (APP (VAR "h",VAR "n"),
                FN
                  ("e1",
                   COND
                     (APP (EQ,RECORD [VAR "e1",VAR "mult"]),VAR "n",
                      RAISE (VAR "e1"))))),FN ("m",APP (VAR "g",VAR "m")),
          FN
            ("v",
             HANDLE
               (RAISE (VAR "mult"),
                FN
                  ("e2",
                   COND
                     (APP (EQ,RECORD [VAR "e2",VAR "ovfl"]),VAR "v",
                      RAISE (VAR "e2")))))],APP (VAR "f",INT 17))),
    APP (MAKEREF,STRING "MUL"))
```

CPS code

```
FIX
   ([("r1",["x2"],APP (VAR "initialNormalCont",[VAR "x2"])),

     ("r1'",["z3"],APP (VAR "initialAbnormalCont",[VAR "z3"])),

     ("F4",["mult","k5"],
        APP (VAR "f",[a.INT 17,VAR "r23",VAR "mult"])),

     ("f",["n","w7","mult"],
        APP (VAR "h",[VAR "n",VAR "r13",VAR "mult"])),

     ("H9",["e1","mult","v"],
        PRIMOP(equal,[VAR "e1",VAR "mult"],[],
           [APP (VAR "F10",[a.INT 0,VAR "v"]),
            APP (VAR "F10",[a.INT 1,VAR "v"])])),

     ("F10",["z11","n"],
        PRIMOP(equal,[VAR "z11",a.INT 0],[],
           [APP (VAR "r23",[VAR "n"]),
            APP (VAR "w7",[VAR "e1"])])),
     ("r13",["x14"],APP (VAR "w7",[VAR "x14"])),

     ("r13'",["z15","mult","v"],APP (VAR "H9",[VAR "z15",VAR "mult",VAR "v"])),

     ("h",["m","w7","mult"],APP (VAR "g",[VAR "m",VAR "r16",VAR "mult"])),

     ("r16",["x17"],APP (VAR "w7",[VAR "x17"])),

     ("r16'",["z18","mult","v"],APP (VAR "r13'",[VAR "z18",VAR "mult",VAR "v"])),

     ("g",["v","w7","mult"],APP (VAR "H19",[VAR "mult",VAR "w7",VAR "v",VAR "mult"])),

     ("H19",["e2","w7","v","mult"],
         PRIMOP(equal,[VAR "e2",VAR "ovfl"],[],
            [APP (VAR "F20",[a.INT 0,VAR "w7",VAR "v",VAR "e2",VAR "mult"]),
             APP (VAR "F20",[a.INT 1,VAR "w7",VAR "v",VAR "e2",VAR "mult"])])),

     ("F20",["z21","w7","v","e2","mult"],
         PRIMOP(equal,[VAR "z21",a.INT 0],[],
            [APP (VAR "w7",[VAR "v",VAR "mult"]),
             APP (VAR "r16'",[VAR "e2",VAR "mult",VAR "v"])])),

     ("r23",["x24"],APP (VAR "r1",[VAR "x24"])),

     ("r23'",["z25"],APP (VAR "ak6",[VAR "z25"]))],

     PRIMOP(makeref,[a.STRING "MUL"],["w26"],
         [APP (VAR "F4",[VAR "w26",VAR "r1"])]))
```

Program 2 contains two functions: **f** and **run**. At the beginning function run is called passing a value of zero as a parameter. Function **run** loops 10 times, calling the same number of times function **f** and making a computation. On the other hand, function **f** computes the expression **n*n** (actually 17*17). The result of the computation never raises an exception so the handler **ovfl** is never evaluated. The program was compiled by using the original (old) approach in the translation of lambda expressions to CPS code.

We showed before that this program produces exception handling overhead (see chapter 5).

PROGRAM 2 - Old technique with exception handler

SML code

```
let
        fun f(n)=n*n handle ovfl=>n
        fun run(x)=if x>10 then f(17) else (run(x+f(17)-288))
in
        run(0)
end
```

Lambda code

```
FIX
  (["f","run"],
   [FN
     ("n",
      HANDLE
        (APP (MULT,RECORD [VAR "n",VAR "n"]),
         FN
           ("e",
            COND
              (APP (EQ,RECORD [VAR "e",VAR "ovfl"]),VAR "n",
               RAISE (VAR "e"))))),
    FN
      ("x",
       COND
         (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
          APP
            (VAR "run",
             APP
               (PLUS,
                RECORD
                  [VAR "x",
                   APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])]))))],
   APP (VAR "run",INT 0))
```

CPS code

```
FIX
  ([("f",["n","w1","x"],
     PRIMOP(gethdlr,[],["h2"],
       [PRIMOP(sethdlr,[VAR "n11"],[],
         [PRIMOP(a.*,[VAR "n",VAR "n"],["w12"],
           [PRIMOP(sethdlr,[VAR "h2"],[],
             [APP (VAR "k4",[VAR "w12",VAR "w1",VAR "x"])])])])])),

   ("k4",["x13","w1","x"],APP (VAR "w1",[VAR "x13",VAR "x"])),

   ("n11",["e3"],
     PRIMOP(sethdlr,[VAR "h2"],[],
       [APP (VAR "F5",[VAR "e3",VAR "k4"])])),

   ("F5",["e","k6"],
     PRIMOP(equal,[VAR "e",VAR "ovfl"],[],
       [APP (VAR "F7",[a.INT 0]),
        APP (VAR "F7",[a.INT 1])])),

   ("F7",["z8"],
     PRIMOP(equal,[VAR "z8",a.INT 0],[],
       [APP (VAR "k6",[VAR "n"]),
        PRIMOP(gethdlr,[],["h9"],
          [APP (VAR "h9",[VAR "e"])])])),

   ("run",["x","w1"],
     PRIMOP(equal,[VAR "x",a.INT 10],[],
       [APP (VAR "F14",[a.INT 0,VAR "x"]),
        APP (VAR "F14",[a.INT 1,VAR "x"])])),

   ("F14",["z15","x"],
     PRIMOP(equal,[VAR "z15",a.INT 0],[],
       [APP (VAR "f",[a.INT 17,VAR "r16",VAR "x"]),
        APP (VAR "f",[a.INT 17,VAR "r22",VAR "x"])])),

   ("r16",["x17","x"],APP (VAR "r25",[VAR "x17"])),

   ("r18",["x19"],APP (VAR "w1",[VAR "x19"])),

   ("r22",["x23","x"],
     PRIMOP(a.-,[VAR "x23",a.INT 288],["w21"],
       [PRIMOP(a.+,[VAR "x",VAR "w21"],["w20"],
         [APP (VAR "run",[VAR "w20",VAR "r18"])])])),

   ("r25",["x26"],APP (VAR "initialNormalCont",[VAR "x26"]))],

   APP (VAR "run",[a.INT 0,VAR "r25"]))
```

In next version of program 2, we eliminate the exception handler from function **f**, and then we compile it again using the original approach in the translation of lambda expressions to CPS code.

PROGRAM 2 - Old technique without exception handler

SML code

```
let
        fun f(n)=n*n
        fun run(x)=if x>1000 then f(17) else (run(x+f(17)-288))
in
        run(0)
end
```

Lambda code

```
FIX
   (["f","run"],
    [FN ("n",APP (MULT,RECORD [VAR "n",VAR "n"])),
     FN
       ("x",
        COND
          (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
           APP
             (VAR "run",
              APP
                (PLUS,
                 RECORD
                   [VAR "x",
                    APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])]))))],
    APP (VAR "run",INT 0))
```

CPS code

```
FIX
   ([("f",["n","w1","x"],
       PRIMOP (a.*,[VAR "n",VAR "n"],["w2"],
         [APP (VAR "w1",[VAR "w2",VAR "x"])])),

     ("run",["x","w1"],
       PRIMOP(equal,[VAR "x",a.INT 10],[],
         [APP (VAR "F3",[a.INT 0,VAR "x"]),
          APP (VAR "F3",[a.INT 1,VAR "x"])])),

     ("F3",["z4","x"],
       PRIMOP(equal,[VAR "z4",a.INT 0],[],
         [APP (VAR "f",[a.INT 17,VAR "r5",VAR "x"]),
          APP (VAR "f",[a.INT 17,VAR "r11",VAR "x"])])),

     ("r5",["x6","x"],APP (VAR "r14",[VAR "x6"])),

     ("r7",["x8"],APP (VAR "w1",[VAR "x8"])),

     ("r11",["x12","x"],
       PRIMOP(a.-,[VAR "x12",a.INT 288],["w10"],
         [PRIMOP(a.+,[VAR "x",VAR "w10"],["w9"],
           [APP (VAR "run",[VAR "w9",VAR "r7"])])])),

     ("r14",["x15"],APP (VAR "initialNormalCont",[VAR "x15"]))],

     APP (VAR "run",[a.INT 0,VAR "r14"]))
```

Now, program 2 (with no handler) is compiled by using the new approach in the translation of lambda expressions to CPS code. As we can see in the produced CPS code , there are two continuations for normal and abnormal computations. Both continuations are passed as arguments by the different functions.

PROGRAM 2 - Two-continuation technique

SML code

```
let
        fun f(n)=n*n handle ovfl=>n
        fun run(x)=if x>10 then f(17) else (run(x+f(17)-288))
in
        run(0)
end
```

Lambda code

```
FIX
  (["f","run"],
   [FN
     ("n",
      HANDLE
        (APP (MULT,RECORD [VAR "n",VAR "n"]),
         FN
           ("e",
            COND
              (APP (EQ,RECORD [VAR "e",VAR "ovfl"]),VAR "n",
               RAISE (VAR "e"))))),
    FN
     ("x",
      COND
        (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
         APP
           (VAR "run",
            APP
              (PLUS,
               RECORD
                 [VAR "x",
                  APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])]))))],
   APP (VAR "run",INT 0))
```

CPS code

```
FIX
  ([("f",["n","w1","ak2","x"],
     PRIMOP (a.*,[VAR "n",VAR "n"],["w7"],
       [APP (VAR "w1",[VAR "w7",VAR "x"])])),

    ("H3",["e"],
     PRIMOP(equal,[VAR "e",VAR "ovfl"],[],
       [APP (VAR "F4",[a.INT 0]),
        APP (VAR "F4",[a.INT 1])])),

    ("F4",["z5"],
     PRIMOP(equal,[VAR "z5",a.INT 0],[],
       [APP (VAR "w1",[VAR "n"]),
        APP (VAR "ak2",[VAR "e"])])),

    ("run",["x","w1","ak2"],
     PRIMOP(equal,[VAR "x",a.INT 10],[],
       [APP (VAR "F8",[a.INT 0,VAR "x"]),
        APP (VAR "F8",[a.INT 1,VAR "x"])])),

    ("F8",["z9","x"],
     PRIMOP(equal,[VAR "z9",a.INT 0],[],
       [APP (VAR "f",[a.INT 17,VAR "r10",VAR "e'12",VAR "x"]),
        APP (VAR "f",[a.INT 17,VAR "r20",VAR "e'22",VAR "x"])])),

    ("r10",["x11","x"],APP (VAR "r25",[VAR "x11"])),

    ("e'12",["z13"],APP (VAR "ak2",[VAR "z13"])),

    ("r14",["x15"],APP (VAR "w1",[VAR "x15"])),

    ("e'16",["z17"],APP (VAR "ak2",[VAR "z17"])),

    ("r20",["x21","x"],
     PRIMOP(a.-,[VAR "x21",a.INT 288],["w19"],
       [PRIMOP(a.+,[VAR "x",VAR "w19"],["w18"],
          [APP(VAR "run",
              [VAR "w18",VAR "r14",VAR "e'16"])])])),

    ("e'22",["z23"],APP (VAR "ak2",[VAR "z23"])),

    ("r25",["x26"],APP (VAR "initialNormalCont",[VAR "x26"])),

    ("e'27",["z28"],APP (VAR "initialAbnormalCont",[VAR "z28"]))],

    APP (VAR "run",[a.INT 0,VAR "r25",VAR "e'27"]))
```

Last, program 2 is compiled by using the modification to the new approach in the translation of lambda expressions to CPS code. As we mentioned before, names of functions for normal and abnormal consecutive computations are similar, and the abnormal continuation is not passed as a parameter.

PROGRAM 2 - One-continuation-displacement technique

SML code

```
let
        fun f(n)=n*n handle ovfl=>n
        fun run(x)=if x>10 then f(17) else (run(x+f(17)-288))
in
        run(0)
end
```

Lambda code

```
FIX
   (["f","run"],
    [FN
       ("n",
        HANDLE
          (APP (MULT,RECORD [VAR "n",VAR "n"]),
           FN
             ("e",
              COND
                (APP (EQ,RECORD [VAR "e",VAR "ovfl"]),VAR "n",
                 RAISE (VAR "e"))))),
     FN
       ("x",
        COND
          (APP (EQ,RECORD [VAR "x",INT 10]),APP (VAR "f",INT 17),
           APP
             (VAR "run",
              APP
                (PLUS,
                 RECORD
                   [VAR "x",
                    APP (SUB,RECORD [APP (VAR "f",INT 17),INT 288])]))))],
    APP (VAR "run",INT 0))
```

CPS code

```
FIX
   ([("f",["n","w1","x"],
       PRIMOP (a.*,[VAR "n",VAR "n"],["w7"],
         [APP (VAR "w1",[VAR "w7",VAR "x"])])),

     ("H3",["e"],
       PRIMOP(equal,[VAR "e",VAR "ovfl"],[],
          [APP (VAR "F4",[a.INT 0]),
           APP (VAR "F4",[a.INT 1])])),

     ("F4",["z5"],
       PRIMOP(equal,[VAR "z5",a.INT 0],[],
          [APP (VAR "w1",[VAR "n"]),
           APP (VAR "w1",[VAR "e"])])),

     ("run",["x","w1"],
      PRIMOP(equal,[VAR "x",a.INT 10],[],
          [APP (VAR "F8",[a.INT 0,VAR "x"]),
           APP (VAR "F8",[a.INT 1,VAR "x"])])),

     ("F8",["z9","x"],
       PRIMOP(equal,[VAR "z9",a.INT 0],[],
          [APP (VAR "f",[a.INT 17,VAR "r10",VAR "x"]),
           APP (VAR "f",[a.INT 17,VAR "r18",VAR "x"])])),

     ("r10",["x11","x"],APP (VAR "r22",[VAR "x11"])),

     ("r10'",["z12"],APP (VAR "ak2",[VAR "z12"])),

     ("r13",["x14"],APP (VAR "w1",[VAR "x14"])),

     ("r13'",["z15"],APP (VAR "ak2",[VAR "z15"])),

     ("r18",["x19","x"],
       PRIMOP(a.-,[VAR "x19",a.INT 288],["w17"],
          [PRIMOP(a.+,[VAR "x",VAR "w17"],["w16"],
             [APP (VAR "run",[VAR "w16",VAR "r13"])])])),

     ("r18'",["z20"],APP (VAR "ak2",[VAR "z20"])),

     ("r22",["x23"],APP (VAR "initialNormalCont",[VAR "x23"])),

     ("r22'",["z24"],APP (VAR "initialAbnormalCont",[VAR "z24"]))],

     APP (VAR "run",[a.INT 0,VAR "r22"]))
```

7.3 Performance evaluation

In the last section, we presented two different programs. We can say that the goal of both programs is different. Program 1 is a program used to test dynamic propagation in the execution of a program. So, the three different versions of program 1 raise one exception in the last called function, which is then caught by the handler defined in the first function. On the other hand, program 2 is a program used to test the performance of the execution of a program. More precisely, it tests the overhead of exception handling in a program. For this case, we present 4 different versions. The first one, is a version of a program that declares and never raises an exception. This version was produced by the compiler that uses the traditional technique of exception handling. This program was presented on chapter 5. The second version is like first version, only with no exception declaration. The third version is again like version one, but using the compiler that implements the two-continuation technique. Finally, the last version is like first version but using the compiler that implements the one-continuation-displacement technique. The next three figures illustrates the performance of program 2 on its four different versions.

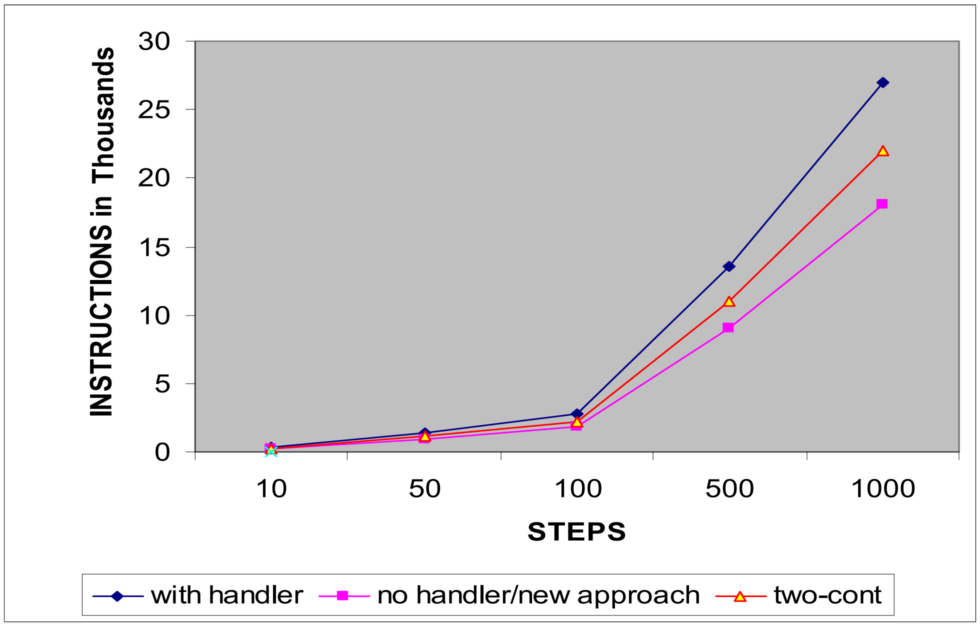

Figure 7.1 Performance of program 2 from 10 to 1000 steps

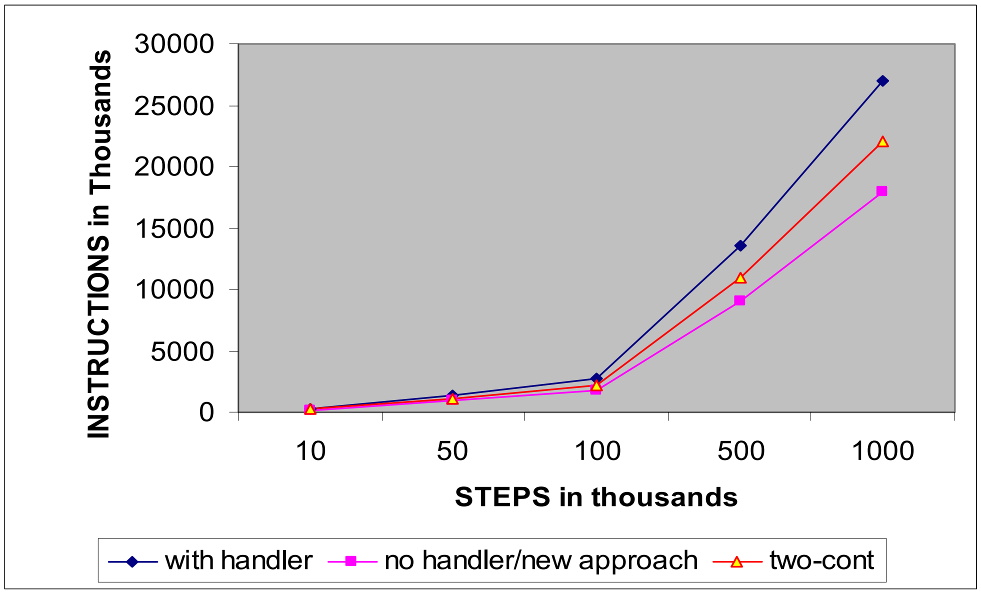

Figure 7.2 Performance of program 2 from 10000 to 1000000 steps

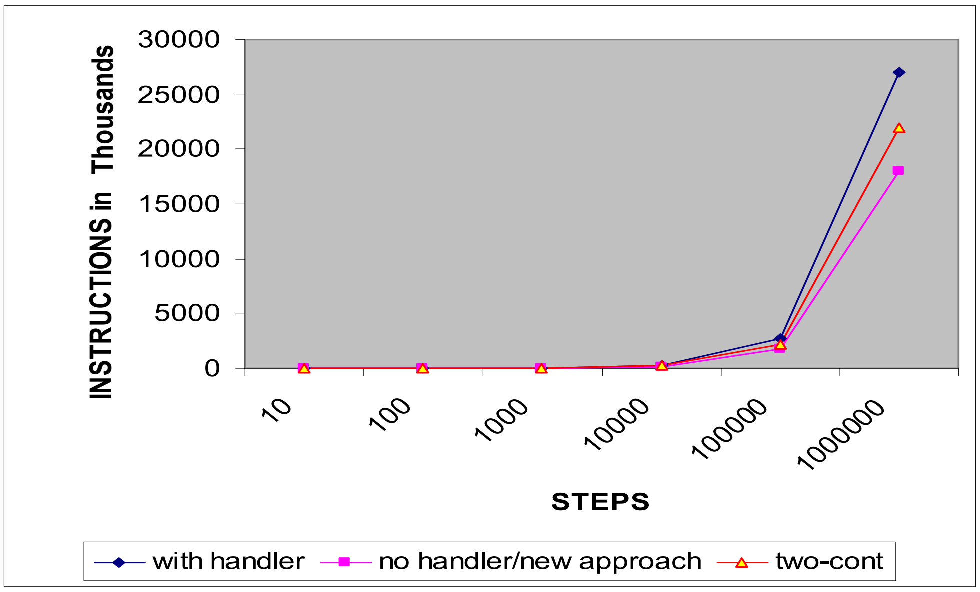

Figure 7.3 Performance of program 2 from 10 to 1000000 steps

7.4 Analysis of performance

Clearly, the last three graphs show three differences in performance of the four versions in program 2. The first version show the worst performance from the four versions. The reason, as we established before, is the overhead created by operators **gethdlr** and **sethdlr**. Because version 2 has no exception declaration, then it has zero overhead exception handling. So, it is the main parameter to compare with relation to the other programs. Version 3 is the program that incorporates the two-continuation technique. The curves of the graphs show that this version decreases the amount of overhead of version 1. Analyzing the code of programs using the two-continuation technique, our conclusion is that the source of overhead is the extra parameter added to each function that passes the normal continuation. Finally, the graphs shows that the curves of version 1 and version 4 are "tied" or follow exactly the same direction. That means, that there is no exception handling overhead in the program that uses the one-continuation-displacement technique.

8 Conclusions and Future Work

8.1 Conclusions

We have implemented a basic CPS compiler for functional languages with exception handling. With it we were able to implement the new approach to exception handling and compare it with the approach taken by the SML of New Jersey compiler. By identifying the source of the overhead we show that all the overhead is moved from the normal flow of control to the code executed when an exception is raised.

The main contributions of this dissertation are:

- We develop a model of translation and execution that allows a programmer (or student/teacher) to write, translates, and executes programs in a source functional language (an extended lambda language) and a target CPS language. The model can be seen as a framework that can be used to execute programs, allowing studying a wide range of performance assessments.
- We implemented an abstract machine. The machine has an instruction set, a register set, a model of memory, a code generator which transforms programs into abstract machine code (AMC), and a simulator which

executes AMC programs. The AMC is essentially an assembly-language program, and like any abstract machine it has some advantages with respect to a real machine: first, we can make very good analysis and experiments of performance, and second it is easy to transport to real architectures.

- We showed that the implementation of exception handling in the SML/NJ and OCAML compilers produces runtime overhead on normal execution.
- We demonstrated the source of the exception handling overhead in SML programs.
- We designed and implemented a new approach where all the overhead is moved from the normal flow of control to the code executed when an exception is raised.

8.2 Future work

The research presented in this dissertation can be extended in the following directions:

- Writing generators of code for real machines. Remember that our generator produces code for an abstract machine.
- Continuing the compiler with the next phase: optimization. We could test our new method of exception handling and see if the code produced by

this method is affected by the optimizer. If this is the case, we could implement some mechanism used in imperative compilers like Ada, Java, or C++ (we discussed some of these techniques in chapter 5).

- Implementing the approach of exception table, or a mix of the two-continuation approach with the exception table approach.
- Our research focused on CPS compilers. Another direction can be working on implementation that use other techniques like combinators. OCAML could be a good example to study.

References

[Ada95] *Ada 95 Reference Manual: The Language, The Standard Libraries,* January 1995. ANSI/ISO/IEC-8652: 1995.

[AM87] Appel, A.W. and MacQueen, D.B. A standard ML compiler. In Gilles Kahn, editor*, Functional Programming Languages and Computer Architecture*, pages 301-324. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, Oregon.

[AM91] Appel, A.W. and MacQueen D.B. Standard ML of New Jersey. *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming, (LNCS 528, Springer-Verlag)* pp. 1-13, August 1991.

[Ans76] *American National Standard Programming Language PL/I.* ANSI X3.53-1976. American National Standards Institute, New Jork.

[App85] Appel, A.W. Semantics-Directed Code Generation. *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages,* January 1985.

[App92] Appel, A.W. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, 1992.

[App98] Appel, A.W. *Modern Compiler Implementation in Java.* Cambridge university press, 1998.

[ASU86] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.

[AT89] Appel, A.W. and Trevor, J. Continuation-Passing, Closure-Passing Style. *Proceedings of the 16th ACM Symposium on Principles on Programming Languages*, ACM, New York, 1989, pp. 293-302.

[Aug84] Augustsson, L. A Compiler for Lazy ML. *ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984.

[Boq99] Boquist, U. Code Optimization Techniques for Lazy Functional Languages. *PhD Dissertation*, Chalmers University of Technology, Gothenburg, Sweden, April 1999.

[BR86] Baker, T.P. and Riccardi, G. A. Implementing Ada Exceptions. *IEEE software*, September 1986, 42-51.

[Car84] Cardelli, L. Compiling a Functional Language. *Proceedings of the 1984 Symposium on Lisp and Functional Programming*. ACM, New York, 1984, pp. 208-226.

[CDGJKN] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 Report (revised). Digital System Research Center, Palo Alto, CA., 1989.

[Cur93] Curien, P.L. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Second edition, CNRS-LIENS, 1993.

[DF98] Douence R., and Fradet, P. A Systematic Study of Functional Language Implementations, *ACM Transactions on Programming Languages and Systems*. Vol. 20, No.2, March 1998, Pages 344-387.

[DGL] Drew, S., Gough, K.J. and Ledermann, J. Implementing Zero Overhead Exception Handling. *technical report*. 95-12, Faculty of Information Technology, Queensland U. of technology, Brisbane, Australia.

[Din00] Dinechin, C. de. C++ Exception Handling. *IEEE Concurrency*, Vol. 8, No.4, October-December 2000, Pages 72-79.

[Fai82] Fairbairn, J. *Ponder and its Type System*, University of Cambridge Computer Laboratory Technical Report No. 31, November 1982.

[FH88] Field, A.J. and Harrison, P.G. *Functional Programming*. Addison-Wesley, 1988.

[FL91] Fradet, P. and Le Metayer, D. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Languages and Systems*. Vol.13, No.1, January 1991, Pages 21-51.

[FOL] FOLDOC Free On-line Dictionary of Computing. Available: http://wombat.doc.ic.ac.uk/foldoc Retrieved: february 27, 2003.

[FWH92] Friedman, D. P., Wand, M. and Haynes, Ch. T. E*ssentials of Programming Languages*. McGraw-Hill, 1992, Chapters 8-10.

[GJS96] Gosling, J., Joy, B. and Steele, G.L. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, Massachusetts, 1996.

[Goo75] Goodenough, J.B. Exception Handling: Issues and a Proposed Notation. *Communications ACM* 18, 683-696, 1975.

[Har98] Harper, R. Programming in Standard ML (notes). Available: *http://www-2.cs.cmu.edu/People/rwh/introsml/* Retrieved: January 10, 2003.

[Hen80] Henderson, P. F*unctional Programming, Applications and Implementation*. Prentice-Hall, 1980.

[HC95] Hilzer Jr., R. C. and Crowl, L.A. A Survey of Sequential and Parallel Implementation Techniques for Functional Programming Languages. Avalilable: *http://www.cs.orst.edu/ ~crowl/paper/ reports /1995R-ORSTCS-95-60-05* Retrieved: January 10, 2003.

[Hud90] Hudak, P. R*eport on the Programming Language HASKELL.* Technical Report YALEU/DCS/RR-777. Yale University, CS Dept., 1990.

[KH89] Kelsey, R., and Hudak, P. Realistic Compilation by Program Transformation. *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, ACM, New York, 1989, pp. 281-292

[KS90] Koenig, A. and Stroustrup, B. Exception Handling for C++. In *USENIX C++,* pages 149-176, April 1990.

[Lan64] Landin, P.J. The Mechanical Evaluation of Expressions. C*omputer Journal*, Vol.6, No. 4, 1964, Pages 308-320.

[Ler00] Leroy, X. *The Objective CAML System:* Documentation and User's Manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available: *http: //caml.inria.fr.* Retrieved: January 10, 2003.

[LS79] Liskov, B., and Snyder, A. Exception Handling in CLU. *IEEE Trans. Software Eng.* 5, 6 (1979), 546-558.

[LS98] Lang, J. and Stewart, D.B.. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology *ACM transactions on programming languages and systems*, Vol. 20, No. 2, March 1998, pages 274-301.

[LYKPMEA] Lee, S., Yang, B., Kim, S., Park, S., Moon, S., Ebcioglu, K. and Altman E. Efficient Java Exception Handling in Just-in-Time Compilation. Available: http://latte.snu.ac.kr/publications/exception_java00_letter.pdf Retrieved: January 10, 2003.

[MS86] Mauny, M. and Suarez, A. Implementing Functional Languages in the Categorical Abstract Machine. *Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM, New York, 1986, pp. 266-278

[MTH90] Milner, R., Tofte, M., and Harper Jr., R.W. *The Definition of Standard ML*. MIT Press, Cambridge, Massachussetts, 1990.

[Nel91] Nelson, G. editor. *Systems Programming with Modula-3*. Prentice Hall series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[OKN01] Ogasawara, T., Komatsu, H. and Nakatani, T. A Study of Exception Handling and its Dynamic Optimization in Java. OOPSLA 2001. Tampa Bay, FL.

[Pau91] Paulson, L.C. *ML for the Working Programmer*. Cambridge University Press, 1991.

[PRHM99] Peyton Jones S., Reid A., Hoare T. and Marlow, S., A Semantic for Imprecise Exceptions. *Proceedings of the SYGPLAN Symposium on Programming Language Design and Implementation (PLDI'99)*, Atlanta, Georgia, 1999.

[PL92] Peyton Jones, S. and Lester, D. *Implementing Functional Languages: a Tutorial*. Prentice-Hall, 1992.

[Rea89] Reade, Ch. *Elements of Functional Languages*. Addison-Wesley, 1989.

[RP00] Ramsey, N. and Peyton Jones, S. A Single Intermediate Language that Supports Multiple Implementations of Exceptions. *PLDI 2000*, Vancouver, British Clumbia, Canada.

[Seb99] Sebesta, R.W. *Concepts of Programming Languages.* Addison-Wesley, fourth edition, 1999.

[Sha94] Shao, Z. Compiling Standard ML for Efficient Execution on Modern Machines. *PhD Dissertation*, Princeton university, 1994.

[Sha97] Shao, Z.. An Overview of the FLINT/ML Ccompiler. *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97).* Amsterdam, The Netherlands, June 1997.

[SML03] Standard SML of New Jersey. Available: *http://www-2.cs.cmu.edu/People/rwh/introsml/* Retrieved: January 10, 2003.

[SML03b] Standard SML of New Jersey: Compilation Manager (CM). Available: *http://cm.bell-labs.com/cm/cs/what/smlnj/doc/CM/index.html* Retrieved: January 10,2003.

[SML03c] Standard SML of New Jersey: The Compiler Structure. Available: *http://cm.bell-labs.com/cm/cs/what/smlnj/doc/Compiler/pages/compiler.html* Retrieved: January 10, 2003.

[SSJ78] Steele, G.L. Jr., and Sussman, G.J. *The Revised Report on SCHEME.* Artificial Intelligence Memo 452. Massachusetts institute of Technology, Cambridge, MA, 1978.

[Sta95] Stansifer, R. *The Study of Programming Languages.* Prentice-Hall, 1995.

[Sta03] Stansifer, R. Exception Handling in Java. Available: *http://cs.fit.edu/~ryan/java/language/exception.html* Retrieved: January 10, 2003.

[Sta03b] Stansifer, R. cse 4250/cse 5250: Programming Languages. Available: *http://cs.fit.edu/~ryan/cse4250* Retrieved: January 10, 2003.

[Ste84] Steele, G. L. Jr. *Common LISP.* Digital Press, Burlington, MA, 1984.

[Str91] Stroustrup, B. The C++ Programming Language. 2d. ed. Addison-Wesley, Reading, MA, 1991.

[Tur79] Turner, D.A. A New Iimplementation Technique for Applicative Languages. *Software-Practice and Experience.* Vol. 9, (1979), 31-49.

[Tur85] Turner, D.A. Miranda: A Non Strict Functional Language with Polymorphic Types. *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architectures*, Lecture Notes in Computer Science (vol. 201), Springer-Verlag, Nancy, France, September 1985.

[Ull98] Ullman, J.D. *Elements of ML Programming*. Prentice-Hall, 1998.

[Ven99] Venners, B. Inside the Java 2 Virtual Machine. McGraw-Hill, second edition, 1999.

[WM95] Wilhelm, R. and Maurer, D. *Compiler Design*. Addison-Wesley, 1995.

[Yal] Yale University Department of Computer Science. THE FLINT PROJECT. Available: http://Flint.cs.yale.edu. Retrieved: January 10, 2003.

[ZS03] Zatarain, R. and Stansifer, R. *Exception Handling for CPS Compilers*. ACMSE 2003,….

Appendix

PROGRAM 1 - Old technique

Abstract Machine Code

```
0           STORE   String MUL,Mem w35
1           LOAD    Mem w35,Reg 1
2           LOAD    Mem r1,Reg 2
3           JUMP    Name F3
      LAB r1:
4           STORE   Reg 1,Mem x2
5           LOAD    Mem x2,Reg 1
6           JUMP    Mem initialNormalCont
      LAB F3:
7           STORE   Reg 1,Mem mult
8           STORE   Reg 2,Mem k4
9           LOAD    Const 17,Reg 1
10          LOAD    Mem r33,Reg 2
11          LOAD    Mem mult,Reg 3
12          JUMP    Name f
      LAB f:
13          STORE   Reg 1,Mem n
14          STORE   Reg 2,Mem w5
15          STORE   Reg 3,Mem mult
16          STORE   Reg 99,Mem h6
17          LOAD    String n15,Reg 99
18          LOAD    Mem n,Reg 1
19          LOAD    Mem r16,Reg 2
20          LOAD    Mem mult,Reg 3
21          LOAD    Mem h6,Reg 4
22          JUMP    Name h
      LAB k8:
23          STORE   Reg 1,Mem x18
24          LOAD    Mem x18,Reg 1
25          JUMP    Name r33
      LAB n15:
26          STORE   Reg 1,Mem e7
27          STORE   Reg 2,Mem h6
28          STORE   Reg 3,Mem v
29          STORE   Reg 4,Mem mult
30          LOAD    String h6,Reg 99
31          LOAD    Mem e7,Reg 1
32          LOAD    Mem k8,Reg 2
33          LOAD    Mem mult,Reg 3
34          LOAD    Mem v,Reg 4
```

```
35          JUMP    Name F9
       LAB F9:
36          STORE   Reg 1,Mem e1
37          STORE   Reg 2,Mem k10
38          STORE   Reg 3,Mem mult
39          STORE   Reg 4,Mem n
40          CJUMP   EQ,Mem e1,Mem mult,L0,L1
       LAB L0:
41          LOAD    Const 0,Reg 1
42          LOAD    Mem n,Reg 2
43          JUMP    Name F11
       LAB L1:
44          LOAD    Const 1,Reg 1
45          LOAD    Mem n,Reg 2
46          JUMP    Name F11
       LAB F11:
47          STORE   Reg 1,Mem z12
48          STORE   Reg 2,Mem n
49          CJUMP   EQ,Mem z12,Const 0,L2,L3
       LAB L2:
50          LOAD    Mem n,Reg 1
51          JUMP    Name k8
       LAB L3:
52          STORE   Reg 99,Mem h13
53          LOAD    Mem e1,Reg 1
54          JUMP    Mem h13
       LAB r16:
55          STORE   Reg 1,Mem x17
56          LOAD    String h6,Reg 99
57          LOAD    Mem x17,Reg 1
58          JUMP    Name k8
       LAB h:
59          STORE   Reg 1,Mem m
60          STORE   Reg 2,Mem w5
61          STORE   Reg 3,Mem mult
62          STORE   Reg 4,Mem h6
63          LOAD    Mem m,Reg 1
64          LOAD    Mem r19,Reg 2
65          LOAD    Mem mult,Reg 3
66          LOAD    Mem h6,Reg 4
67          JUMP    Name g
       LAB r19:
68          STORE   Reg 1,Mem x20
69          LOAD    Mem x20,Reg 1
70          JUMP    Mem w5
       LAB g:
71          STORE   Reg 1,Mem v
72          STORE   Reg 2,Mem w5
73          STORE   Reg 3,Mem mult
74          STORE   Reg 4,Mem h6
75          STORE   Reg 99,Mem h21
76          LOAD    String n30,Reg 99
```

```
77          STORE   Reg 99,Mem h31
78          LOAD    Mem mult,Reg 1
79          LOAD    Mem h21,Reg 2
80          LOAD    Mem h6,Reg 3
81          LOAD    Mem v,Reg 4
82          LOAD    Mem mult,Reg 5
83          JUMP    Mem h31
     LAB k23:
84          STORE   Reg 1,Mem x32
85          LOAD    Mem x32,Reg 1
86          JUMP    Mem w5
     LAB n30:
87          STORE   Reg 1,Mem e22
88          STORE   Reg 2,Mem h21
89          STORE   Reg 3,Mem h6
90          STORE   Reg 4,Mem v
91          STORE   Reg 5,Mem mult
92          LOAD    String h21,Reg 99
93          LOAD    Mem e22,Reg 1
94          LOAD    Mem k23,Reg 2
95          LOAD    Mem h6,Reg 3
96          LOAD    Mem v,Reg 4
97          LOAD    Mem mult,Reg 5
98          JUMP    Name F24
     LAB F24:
99          STORE   Reg 1,Mem e2
100         STORE   Reg 2,Mem k25
101         STORE   Reg 3,Mem h6
102         STORE   Reg 4,Mem v
103         STORE   Reg 5,Mem mult
104         CJUMP   EQ,Mem e2,Mem ovfl,L4,L5
     LAB L4:
105         LOAD    Const 0,Reg 1
106         LOAD    Mem e2,Reg 2
107         LOAD    Mem h6,Reg 3
108         LOAD    Mem v,Reg 4
109         LOAD    Mem mult,Reg 5
110         JUMP    Name F26
     LAB L5:
111         LOAD    Const 1,Reg 1
112         LOAD    Mem e2,Reg 2
113         LOAD    Mem h6,Reg 3
114         LOAD    Mem v,Reg 4
115         LOAD    Mem mult,Reg 5
116         JUMP    Name F26
     LAB F26:
117         STORE   Reg 1,Mem z27
118         STORE   Reg 2,Mem e2
119         STORE   Reg 3,Mem h6
120         STORE   Reg 4,Mem v
121         STORE   Reg 5,Mem mult
122         CJUMP   EQ,Mem z27,Const 0,L6,L7
```

```
        LAB L6:
123              LOAD    Mem v,Reg 1
124              JUMP    Mem k25
        LAB L7:
125              STORE   Reg 99,Mem h28
126              LOAD    Mem e2,Reg 1
127              LOAD    Mem h6,Reg 2
128              LOAD    Mem v,Reg 3
129              LOAD    Mem mult,Reg 4
130              JUMP    Mem h28
        LAB r33:
131              STORE   Reg 1,Mem x34
132              LOAD    Mem x34,Reg 1
133              JUMP    Name r1
        LAB end:
```

PROGRAM 1- Two-continuation technique

Abstract Machine Code

```
0          STORE   String MUL,Mem w30
1          LOAD    Mem w30,Reg 1
2          LOAD    Mem r1,Reg 2
3          LOAD    Mem e'3,Reg 3
4          JUMP    Name F5
     LAB r1:
5          STORE   Reg 1,Mem x2
6          LOAD    Mem x2,Reg 1
7          JUMP    Mem initialNormalCont
     LAB e'3:
8          STORE   Reg 1,Mem z4
9          LOAD    Mem z4,Reg 1
10         JUMP    Mem initialAbnormalCont
     LAB F5:
11         STORE   Reg 1,Mem mult
12         STORE   Reg 2,Mem k6
13         STORE   Reg 3,Mem ak7
14         LOAD    Const 17,Reg 1
15         LOAD    Mem r26,Reg 2
16         LOAD    Mem e'28,Reg 3
17         LOAD    Mem mult,Reg 4
18         JUMP    Name f
     LAB f:
19         STORE   Reg 1,Mem n
20         STORE   Reg 2,Mem w8
21         STORE   Reg 3,Mem ak9
22         STORE   Reg 4,Mem mult
23         LOAD    Mem n,Reg 1
24         LOAD    Mem r14,Reg 2
25         LOAD    Mem e'16,Reg 3
26         LOAD    Mem mult,Reg 4
27         JUMP    Name h
     LAB H10:
28         STORE   Reg 1,Mem e1
29         STORE   Reg 2,Mem mult
30         STORE   Reg 3,Mem v
31         CJUMP   EQ,Mem e1,Mem mult,L0,L1
     LAB L0:
32         LOAD    Const 0,Reg 1
33         LOAD    Mem v,Reg 2
34         JUMP    Name F11
     LAB L1:
35         LOAD    Const 1,Reg 1
36         LOAD    Mem v,Reg 2
37         JUMP    Name F11
     LAB F11:
38         STORE   Reg 1,Mem z12
```

```
39          STORE   Reg 2,Mem n
40          CJUMP   EQ,Mem z12,Const 0,L2,L3
     LAB L2:
41          LOAD    Mem n,Reg 1
42          JUMP    Name r26
     LAB L3:
43          LOAD    Mem e1,Reg 1
44          JUMP    Mem ak9
     LAB r14:
45          STORE   Reg 1,Mem x15
46          LOAD    Mem x15,Reg 1
47          JUMP    Mem w8
     LAB e'16:
48          STORE   Reg 1,Mem z17
49          STORE   Reg 2,Mem mult
50          STORE   Reg 3,Mem v
51          LOAD    Mem z17,Reg 1
52          LOAD    Mem mult,Reg 2
53          LOAD    Mem v,Reg 3
54          JUMP    Name H10
     LAB h:
55          STORE   Reg 1,Mem m
56          STORE   Reg 2,Mem w8
57          STORE   Reg 3,Mem ak9
58          STORE   Reg 4,Mem mult
59          LOAD    Mem m,Reg 1
60          LOAD    Mem r18,Reg 2
61          LOAD    Mem e'20,Reg 3
62          LOAD    Mem mult,Reg 4
63          JUMP    Name g
     LAB r18:
64          STORE   Reg 1,Mem x19
65          LOAD    Mem x19,Reg 1
66          JUMP    Mem w8
     LAB e'20:
67          STORE   Reg 1,Mem z21
68          STORE   Reg 2,Mem mult
69          STORE   Reg 3,Mem v
70          LOAD    Mem z21,Reg 1
71          LOAD    Mem mult,Reg 2
72          LOAD    Mem v,Reg 3
73          JUMP    Name e'16
     LAB g:
74          STORE   Reg 1,Mem v
75          STORE   Reg 2,Mem w8
76          STORE   Reg 3,Mem ak9
77          STORE   Reg 4,Mem mult
78          LOAD    Mem mult,Reg 1
79          LOAD    Mem w8,Reg 2
80          LOAD    Mem v,Reg 3
81          LOAD    Mem mult,Reg 4
82          JUMP    Name H22
```

```
        LAB H22:
83              STORE   Reg 1,Mem e2
84              STORE   Reg 2,Mem w8
85              STORE   Reg 3,Mem v
86              STORE   Reg 4,Mem mult
87              CJUMP   EQ,Mem e2,Mem ovfl,L4,L5
        LAB L4:
88              LOAD    Const 0,Reg 1
89              LOAD    Mem w8,Reg 2
90              LOAD    Mem v,Reg 3
91              LOAD    Mem e2,Reg 4
92              LOAD    Mem mult,Reg 5
93              JUMP    Name F23
        LAB L5:
94              LOAD    Const 1,Reg 1
95              LOAD    Mem w8,Reg 2
96              LOAD    Mem v,Reg 3
97              LOAD    Mem e2,Reg 4
98              LOAD    Mem mult,Reg 5
99              JUMP    Name F23
        LAB F23:
100             STORE   Reg 1,Mem z24
101             STORE   Reg 2,Mem w8
102             STORE   Reg 3,Mem v
103             STORE   Reg 4,Mem e2
104             STORE   Reg 5,Mem mult
105             CJUMP   EQ,Mem z24,Const 0,L6,L7
        LAB L6:
106             LOAD    Mem v,Reg 1
107             LOAD    Mem mult,Reg 2
108             JUMP    Mem w8
        LAB L7:
109             LOAD    Mem e2,Reg 1
110             LOAD    Mem mult,Reg 2
111             LOAD    Mem v,Reg 3
112             JUMP    Name e'20
        LAB r26:
113             STORE   Reg 1,Mem x27
114             LOAD    Mem x27,Reg 1
115             JUMP    Name r1
        LAB e'28:
116             STORE   Reg 1,Mem z29
117             LOAD    Mem z29,Reg 1
118             JUMP    Mem ak7
        LAB end:
```

PROGRAM 1- One-continuation-displacement technique

Abstract Machine Code

```
0          STORE   String MUL,Mem w26
1          LOAD    Mem w26,Reg 1
2          LOAD    Mem r1,Reg 2
3          JUMP    Name F4
     LAB r1:
4          STORE   Reg 1,Mem x2
5          LOAD    Mem x2,Reg 1
6          JUMP    Mem initialNormalCont
     LAB r1':
7          STORE   Reg 1,Mem z3
8          LOAD    Mem z3,Reg 1
9          JUMP    Mem initialAbnormalCont
     LAB F4:
10         STORE   Reg 1,Mem mult
11         STORE   Reg 2,Mem k5
12         LOAD    Const 17,Reg 1
13         LOAD    Mem r23,Reg 2
14         LOAD    Mem mult,Reg 3
15         JUMP    Name f
     LAB f:
16         STORE   Reg 1,Mem n
17         STORE   Reg 2,Mem w7
18         STORE   Reg 3,Mem mult
19         LOAD    Mem n,Reg 1
20         LOAD    Mem r13,Reg 2
21         LOAD    Mem mult,Reg 3
22         JUMP    Name h
     LAB H9:
23         STORE   Reg 1,Mem e1
24         STORE   Reg 2,Mem mult
25         STORE   Reg 3,Mem v
26         CJUMP   EQ,Mem e1,Mem mult,L0,L1
     LAB L0:
27         LOAD    Const 0,Reg 1
28         LOAD    Mem v,Reg 2
29         JUMP    Name F10
     LAB L1:
30         LOAD    Const 1,Reg 1
31         LOAD    Mem v,Reg 2
32         JUMP    Name F10
     LAB F10:
33         STORE   Reg 1,Mem z11
34         STORE   Reg 2,Mem n
35         CJUMP   EQ,Mem z11,Const 0,L2,L3
     LAB L2:
36         LOAD    Mem n,Reg 1
37         JUMP    Name r23
```

```
     LAB L3:
38          LOAD    Mem e1,Reg 1
39          JUMP    Mem w7
     LAB r13:
40          STORE   Reg 1,Mem x14
41          LOAD    Mem x14,Reg 1
42          JUMP    Mem w7
     LAB r13':
43          STORE   Reg 1,Mem z15
44          STORE   Reg 2,Mem mult
45          STORE   Reg 3,Mem v
46          LOAD    Mem z15,Reg 1
47          LOAD    Mem mult,Reg 2
48          LOAD    Mem v,Reg 3
49          JUMP    Name H9
     LAB h:
50          STORE   Reg 1,Mem m
51          STORE   Reg 2,Mem w7
52          STORE   Reg 3,Mem mult
53          LOAD    Mem m,Reg 1
54          LOAD    Mem r16,Reg 2
55          LOAD    Mem mult,Reg 3
56          JUMP    Name g
     LAB r16:
57          STORE   Reg 1,Mem x17
58          LOAD    Mem x17,Reg 1
59          JUMP    Mem w7
     LAB r16':
60          STORE   Reg 1,Mem z18
61          STORE   Reg 2,Mem mult
62          STORE   Reg 3,Mem v
63          LOAD    Mem z18,Reg 1
64          LOAD    Mem mult,Reg 2
65          LOAD    Mem v,Reg 3
66          JUMP    Name r13'
     LAB g:
67          STORE   Reg 1,Mem v
68          STORE   Reg 2,Mem w7
69          STORE   Reg 3,Mem mult
70          LOAD    Mem mult,Reg 1
71          LOAD    Mem w7,Reg 2
72          LOAD    Mem v,Reg 3
73          LOAD    Mem mult,Reg 4
74          JUMP    Name H19
     LAB H19:
75          STORE   Reg 1,Mem e2
76          STORE   Reg 2,Mem w7
77          STORE   Reg 3,Mem v
78          STORE   Reg 4,Mem mult
79          CJUMP   EQ,Mem e2,Mem ovfl,L4,L5
     LAB L4:
80          LOAD    Const 0,Reg 1
```

```
81          LOAD    Mem w7,Reg 2
82          LOAD    Mem v,Reg 3
83          LOAD    Mem e2,Reg 4
84          LOAD    Mem mult,Reg 5
85          JUMP    Name F20
     LAB L5:
86          LOAD    Const 1,Reg 1
87          LOAD    Mem w7,Reg 2
88          LOAD    Mem v,Reg 3
89          LOAD    Mem e2,Reg 4
90          LOAD    Mem mult,Reg 5
91          JUMP    Name F20
     LAB F20:
92          STORE   Reg 1,Mem z21
93          STORE   Reg 2,Mem w7
94          STORE   Reg 3,Mem v
95          STORE   Reg 4,Mem e2
96          STORE   Reg 5,Mem mult
97          CJUMP   EQ,Mem z21,Const 0,L6,L7
     LAB L6:
98          LOAD    Mem v,Reg 1
99          LOAD    Mem mult,Reg 2
100         JUMP    Mem w7
     LAB L7:
101         LOAD    Mem e2,Reg 1
102         LOAD    Mem mult,Reg 2
103         LOAD    Mem v,Reg 3
104         JUMP    Name r16'
     LAB r23:
105         STORE   Reg 1,Mem x24
106         LOAD    Mem x24,Reg 1
107         JUMP    Name r1
     LAB r23':
108         STORE   Reg 1,Mem z25
109         LOAD    Mem z25,Reg 1
110         JUMP    Mem ak6
     LAB end:
```

PROGRAM 2 - Old technique with exception handler

Abstract Machine Code

```
0          LOAD    Const 0,Reg 1
1          LOAD    Mem r25,Reg 2
2          JUMP    Name run
      LAB f:
3          STORE   Reg 1,Mem n
4          STORE   Reg 2,Mem w1
5          STORE   Reg 3,Mem x
6          STORE   Reg 99,Mem h2
7          LOAD    String n11,Reg 99
8          MUL     Mem n,Mem n,Mem w12
9          LOAD    String h2,Reg 99
10         LOAD    Mem w12,Reg 1
11         LOAD    Mem w1,Reg 2
12         LOAD    Mem x,Reg 3
13         JUMP    Name k4
      LAB k4:
14         STORE   Reg 1,Mem x13
15         STORE   Reg 2,Mem w1
16         STORE   Reg 3,Mem x
17         LOAD    Mem x13,Reg 1
18         LOAD    Mem x,Reg 2
19         JUMP    Mem w1
      LAB n11:
20         STORE   Reg 1,Mem e3
21         LOAD    String h2,Reg 99
22         LOAD    Mem e3,Reg 1
23         LOAD    Mem k4,Reg 2
24         JUMP    Name F5
      LAB F5:
25         STORE   Reg 1,Mem e
26         STORE   Reg 2,Mem k6
27         CJUMP   EQ,Mem e,Mem ovfl,L0,L1
      LAB L0:
28         LOAD    Const 0,Reg 1
29         JUMP    Name F7
      LAB L1:
30         LOAD    Const 1,Reg 1
31         JUMP    Name F7
      LAB F7:
32         STORE   Reg 1,Mem z8
33         CJUMP   EQ,Mem z8,Const 0,L2,L3
      LAB L2:
34         LOAD    Mem n,Reg 1
35         JUMP    Mem k6
      LAB L3:
36         STORE   Reg 99,Mem h9
37         LOAD    Mem e,Reg 1
```

```
38        JUMP    Mem h9
     LAB run:
39        STORE   Reg 1,Mem x
40        STORE   Reg 2,Mem w1
41        CJUMP   EQ,Mem x,Const 10,L4,L5
     LAB L4:
42        LOAD    Const 0,Reg 1
43        LOAD    Mem x,Reg 2
44        JUMP    Name F14
     LAB L5:
45        LOAD    Const 1,Reg 1
46        LOAD    Mem x,Reg 2
47        JUMP    Name F14
     LAB F14:
48        STORE   Reg 1,Mem z15
49        STORE   Reg 2,Mem x
50        CJUMP   EQ,Mem z15,Const 0,L6,L7
     LAB L6:
51        LOAD    Const 17,Reg 1
52        LOAD    Mem r16,Reg 2
53        LOAD    Mem x,Reg 3
54        JUMP    Name f
     LAB L7:
55        LOAD    Const 17,Reg 1
56        LOAD    Mem r22,Reg 2
57        LOAD    Mem x,Reg 3
58        JUMP    Name f
     LAB r16:
59        STORE   Reg 1,Mem x17
60        STORE   Reg 2,Mem x
61        LOAD    Mem x17,Reg 1
62        JUMP    Name r25
     LAB r18:
63        STORE   Reg 1,Mem x19
64        LOAD    Mem x19,Reg 1
65        JUMP    Mem w1
     LAB r22:
66        STORE   Reg 1,Mem x23
67        STORE   Reg 2,Mem x
68        SUB     Mem x23,Const 288,Mem w21
69        ADD     Mem x,Mem w21,Mem w20
70        LOAD    Mem w20,Reg 1
71        LOAD    Mem r18,Reg 2
72        JUMP    Name run
     LAB r25:
73        STORE   Reg 1,Mem x26
74        LOAD    Mem x26,Reg 1
75        JUMP    Mem initialNormalCont
     LAB end:
```

PROGRAM 2 - Old technique without exception handler

Abstract Machine Code

```
0          LOAD    Const 0,Reg 1
1          LOAD    Mem r14,Reg 2
2          JUMP    Name run
      LAB f:
3          STORE   Reg 1,Mem n
4          STORE   Reg 2,Mem w1
5          STORE   Reg 3,Mem x
6          MUL     Mem n,Mem n,Mem w2
7          LOAD    Mem w2,Reg 1
8          LOAD    Mem x,Reg 2
9          JUMP    Mem w1
      LAB run:
10         STORE   Reg 1,Mem x
11         STORE   Reg 2,Mem w1
12         CJUMP   EQ,Mem x,Const 10,L0,L1
      LAB L0:
13         LOAD    Const 0,Reg 1
14         LOAD    Mem x,Reg 2
15         JUMP    Name F3
      LAB L1:
16         LOAD    Const 1,Reg 1
17         LOAD    Mem x,Reg 2
18         JUMP    Name F3
      LAB F3:
19         STORE   Reg 1,Mem z4
20         STORE   Reg 2,Mem x
21         CJUMP   EQ,Mem z4,Const 0,L2,L3
      LAB L2:
22         LOAD    Const 17,Reg 1
23         LOAD    Mem r5,Reg 2
24         LOAD    Mem x,Reg 3
25         JUMP    Name f
      LAB L3:
26         LOAD    Const 17,Reg 1
27         LOAD    Mem r11,Reg 2
28         LOAD    Mem x,Reg 3
29         JUMP    Name f
      LAB r5:
30         STORE   Reg 1,Mem x6
31         STORE   Reg 2,Mem x
32         LOAD    Mem x6,Reg 1
33         JUMP    Name r14
      LAB r7:
34         STORE   Reg 1,Mem x8
35         LOAD    Mem x8,Reg 1
36         JUMP    Mem w1
      LAB r11:
```

```
37          STORE   Reg 1,Mem x12
38          STORE   Reg 2,Mem x
39          SUB     Mem x12,Const 288,Mem w10
40          ADD     Mem x,Mem w10,Mem w9
41          LOAD    Mem w9,Reg 1
42          LOAD    Mem r7,Reg 2
43          JUMP    Name run
      LAB r14:
44          STORE   Reg 1,Mem x15
45          LOAD    Mem x15,Reg 1
46          JUMP    Mem initialNormalCont
      LAB end:
```

PROGRAM 2 - Two-continuation technique

Abstract Machine Code

```
0          LOAD    Const 0,Reg 1
1          LOAD    Mem r25,Reg 2
2          LOAD    Mem e'27,Reg 3
3          JUMP    Name run
    LAB f:
4          STORE   Reg 1,Mem n
5          STORE   Reg 2,Mem w1
6          STORE   Reg 3,Mem ak2
7          STORE   Reg 4,Mem x
8          MUL     Mem n,Mem n,Mem w7
9          LOAD    Mem w7,Reg 1
10         LOAD    Mem x,Reg 2
11         JUMP    Mem w1
    LAB H3:
12         STORE   Reg 1,Mem e
13         CJUMP   EQ,Mem e,Mem ovfl,L0,L1
    LAB L0:
14         LOAD    Const 0,Reg 1
15         JUMP    Name F4
    LAB L1:
16         LOAD    Const 1,Reg 1
17         JUMP    Name F4
    LAB F4:
18         STORE   Reg 1,Mem z5
19         CJUMP   EQ,Mem z5,Const 0,L2,L3
    LAB L2:
20         LOAD    Mem n,Reg 1
21         JUMP    Mem w1
    LAB L3:
22         LOAD    Mem e,Reg 1
23         JUMP    Mem ak2
    LAB run:
24         STORE   Reg 1,Mem x
25         STORE   Reg 2,Mem w1
26         STORE   Reg 3,Mem ak2
27         CJUMP   EQ,Mem x,Const 10,L4,L5
    LAB L4:
28         LOAD    Const 0,Reg 1
29         LOAD    Mem x,Reg 2
30         JUMP    Name F8
    LAB L5:
31         LOAD    Const 1,Reg 1
32         LOAD    Mem x,Reg 2
33         JUMP    Name F8
    LAB F8:
34         STORE   Reg 1,Mem z9
35         STORE   Reg 2,Mem x
```

```
36          CJUMP   EQ,Mem z9,Const 0,L6,L7
       LAB L6:
37          LOAD    Const 17,Reg 1
38          LOAD    Mem r10,Reg 2
39          LOAD    Mem e'12,Reg 3
40          LOAD    Mem x,Reg 4
41          JUMP    Name f
       LAB L7:
42          LOAD    Const 17,Reg 1
43          LOAD    Mem r20,Reg 2
44          LOAD    Mem e'22,Reg 3
45          LOAD    Mem x,Reg 4
46          JUMP    Name f
       LAB r10:
47          STORE   Reg 1,Mem x11
48          STORE   Reg 2,Mem x
49          LOAD    Mem x11,Reg 1
50          JUMP    Name r25
       LAB e'12:
51          STORE   Reg 1,Mem z13
52          LOAD    Mem z13,Reg 1
53          JUMP    Mem ak2
       LAB r14:
54          STORE   Reg 1,Mem x15
55          LOAD    Mem x15,Reg 1
56          JUMP    Mem w1
       LAB e'16:
57          STORE   Reg 1,Mem z17
58          LOAD    Mem z17,Reg 1
59          JUMP    Mem ak2
       LAB r20:
60          STORE   Reg 1,Mem x21
61          STORE   Reg 2,Mem x
62          SUB     Mem x21,Const 288,Mem w19
63          ADD     Mem x,Mem w19,Mem w18
64          LOAD    Mem w18,Reg 1
65          LOAD    Mem r14,Reg 2
66          LOAD    Mem e'16,Reg 3
67          JUMP    Name run
       LAB e'22:
68          STORE   Reg 1,Mem z23
69          LOAD    Mem z23,Reg 1
70          JUMP    Mem ak2
       LAB r25:
71          STORE   Reg 1,Mem x26
72          LOAD    Mem x26,Reg 1
73          JUMP    Mem initialNormalCont
       LAB e'27:
74          STORE   Reg 1,Mem z28
75          LOAD    Mem z28,Reg 1
76          JUMP    Mem initialAbnormalCont
       LAB end:
```

PROGRAM 2 - One-continuation-displacement technique

Abstract Machine Code

```
0           LOAD    Const 0,Reg 1
1           LOAD    Mem r22,Reg 2
2           JUMP    Name run
      LAB f:
3           STORE   Reg 1,Mem n
4           STORE   Reg 2,Mem w1
5           STORE   Reg 3,Mem x
6           MUL     Mem n,Mem n,Mem w7
7           LOAD    Mem w7,Reg 1
8           LOAD    Mem x,Reg 2
9           JUMP    Mem w1
      LAB H3:
10          STORE   Reg 1,Mem e
11          CJUMP   EQ,Mem e,Mem ovfl,L0,L1
      LAB L0:
12          LOAD    Const 0,Reg 1
13          JUMP    Name F4
      LAB L1:
14          LOAD    Const 1,Reg 1
15          JUMP    Name F4
      LAB F4:
16          STORE   Reg 1,Mem z5
17          CJUMP   EQ,Mem z5,Const 0,L2,L3
      LAB L2:
18          LOAD    Mem n,Reg 1
19          JUMP    Mem w1
      LAB L3:
20          LOAD    Mem e,Reg 1
21          JUMP    Mem w1
      LAB run:
22          STORE   Reg 1,Mem x
23          STORE   Reg 2,Mem w1
24          CJUMP   EQ,Mem x,Const 10,L4,L5
      LAB L4:
25          LOAD    Const 0,Reg 1
26          LOAD    Mem x,Reg 2
27          JUMP    Name F8
      LAB L5:
28          LOAD    Const 1,Reg 1
29          LOAD    Mem x,Reg 2
30          JUMP    Name F8
      LAB F8:
31          STORE   Reg 1,Mem z9
32          STORE   Reg 2,Mem x
33          CJUMP   EQ,Mem z9,Const 0,L6,L7
      LAB L6:
```

```
34          LOAD    Const 17,Reg 1
35          LOAD    Mem r10,Reg 2
36          LOAD    Mem x,Reg 3
37          JUMP    Name f
      LAB L7:
38          LOAD    Const 17,Reg 1
39          LOAD    Mem r18,Reg 2
40          LOAD    Mem x,Reg 3
41          JUMP    Name f
      LAB r10:
42          STORE   Reg 1,Mem x11
43          STORE   Reg 2,Mem x
44          LOAD    Mem x11,Reg 1
45          JUMP    Name r22
      LAB r10':
46          STORE   Reg 1,Mem z12
47          LOAD    Mem z12,Reg 1
48          JUMP    Mem ak2
      LAB r13:
49          STORE   Reg 1,Mem x14
50          LOAD    Mem x14,Reg 1
51          JUMP    Mem w1
      LAB r13':
52          STORE   Reg 1,Mem z15
53          LOAD    Mem z15,Reg 1
54          JUMP    Mem ak2
      LAB r18:
55          STORE   Reg 1,Mem x19
56          STORE   Reg 2,Mem x
57          SUB     Mem x19,Const 288,Mem w17
58          ADD     Mem x,Mem w17,Mem w16
59          LOAD    Mem w16,Reg 1
60          LOAD    Mem r13,Reg 2
61          JUMP    Name run
      LAB r18':
62          STORE   Reg 1,Mem z20
63          LOAD    Mem z20,Reg 1
64          JUMP    Mem ak2
      LAB r22:
65          STORE   Reg 1,Mem x23
66          LOAD    Mem x23,Reg 1
67          JUMP    Mem initialNormalCont
      LAB r22':
68          STORE   Reg 1,Mem z24
69          LOAD    Mem z24,Reg 1
70          JUMP    Mem initialAbnormalCont
      LAB end:
```