Implementing a CPS Compiler for Functional Languages with Zero Overhead Exception Handling

Ramón Zatarain Cabada,[1] Ryan Stansifer,[2] and
María Lucía Barrón Estrada[1]

[1]Instituto tecnológico de Culiacán, Av. Juan de Dios Batiz s/n, Col. Guadalupe, Culiacán, Sin. CP 80220 Mexico
rzatarain@itculiacan.edu.mx mbarron@fit.edu
[2]Florida Institute of Technology, 150 W. University Blvd. Melbourne, FL. 30901 USA
ryan@cs.fit.edu

Abstract. We have implemented a basic CPS compiler for functional languages with exception handling. With it we were able to implement a new approach to exception handling and compare it side-by-side with the approach taken by the SML of New Jersey compiler. 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 new approach and an experiment were shown in [9,10] where we concluded that programs with exception handling using our new approach do not produce overhead. In this paper we show more details about the compiler. This includes lambda, CPS and abstract machine code generation. At the end, we show some results of the experiments.

1 Introduction

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 [11]. ML [8] is a general-purpose functional programming language designed for large projects. 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 [3]. The compiler translates a source program into a target machine language program in several phases.
Compilers of imperative languages like Java, Ada, and C++ implement exception handling without imposing overhead on normal execution [4, 5, 6]. 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 or CAML [7] produce code that has exception-handling overhead [10].

2 A Model of CPS Translation

The middle part and key transformation in some functional language compilers is the conversion to CPS (continuation-passing style) language. We use CPS as our intermediate representation in our functional language compiler (figure 1). The compiler first translates a source program written in lambda code into CPS code. Then, it translates the CPS code into a one-function CPS program named flat CPS. Finally, flat CPS code is translated into abstract machine code (AMC).

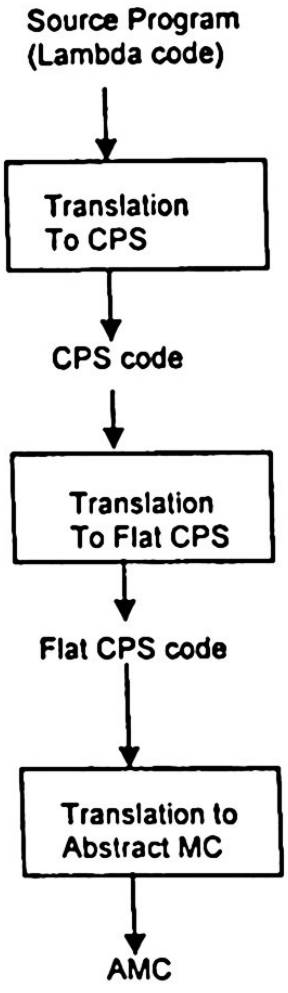

Fig. 1. Overview of the compiler

2.1 The Lambda Language

Our source language is written using lambda code. The next SML code shows the definition of a lambda expression. We start here to avoid issues of parsing a concrete source language.

```
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
```

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** that 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 given exception.

2.2 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 [2]:

- 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.

The next SML code shows the definition of a CPS expression.

```
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
```

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 (specially for declaring an exception).

Datatype **value** 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. Like we established 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.

2.3 The Abstract Machine Code (AMC)

Continuation-passing style is used because it is closely related to Church's lambda calculus and to the model of von Neumann, here represented by our target abstract machine language (see figure 2). Each operator of the CPS corresponds to one opera-

tor 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 2 illustrates the components of the abstract machine.

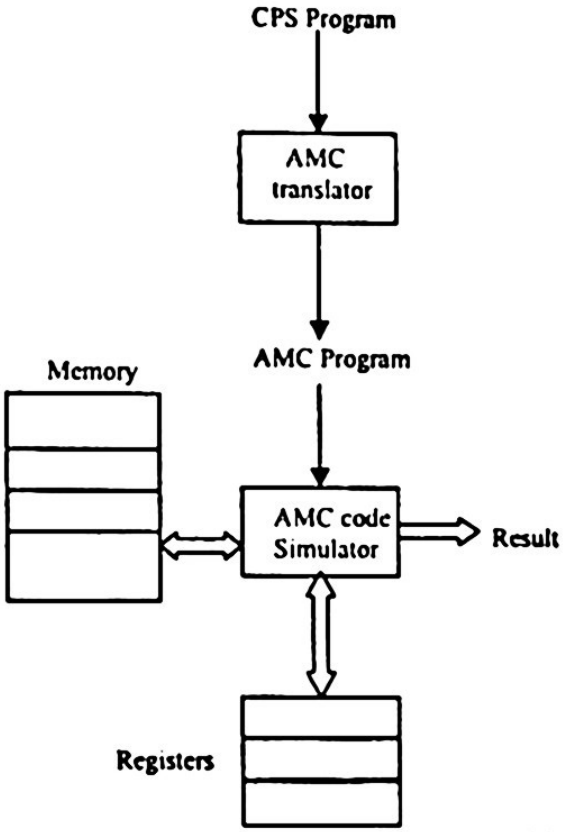

Fig. 2. Components of the abstract machine

During the compilation process, lambda expressions are translated into corresponding CPS expressions. Then, CPS expressions are translated into abstract machine code.
The AMC is essentially an assembly-language code, and like any abstract machine it has some advantages with respect to a real machine: first, creating a simulator for the abstract machine is no big deal; and second making performance analysis is very convenient (we can include code for performance analysis inside the machine). The next SML code shows the definition of the format for AMC instructions and expressions.

```
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
```

An abstract machine instruction can be:

- A label that represents 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 any label.
- A CJUMP is a conditional jump to one of two labels, depending of the result of the test.
- A load 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.

and a expression 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.
- A register (registers have a unique number).

Example:

We illustrate all the different code representations with a complete program in SML, Lambda, 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"])))

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:
```

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 both parameters (constant 4 and register 3) 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**, which is a fixed address or constant in memory that represents the end of any program executed in the abstract machine.

2.4 A Simulator of the AMC

The simulator is a program, which simulates a real computer. It is a piece of software that runs an AMC program. In order to simulate 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 2). 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 used for the simulation of an AMC program.

Input: An AMC program, which is kept as a list of instructions.
Output: A value or result after executing the AMC program.
Method:

- Convert the instructions kept as a list into a different data structure: an array. The array is 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 and memory.

3 A New Approach for Implementing Exception Handling

Implementing exception handling in SML/NJ and OCAML produce runtime overhead [10]. The implementation of our compiler generates also exception-handling overhead [9]. We have implemented a new technique for zero overhead exception handling. A more detailed explanation of the source of that overhead can be found in [9,10].

3.1 Experiments

This section presents one example of the experiments we made in order to test the performance of the new technique.
In section 2.4 we gave an explanation of the implementation of a simulator for a real machine. The simulator "executes" a program in abstract machine code, getting then, 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 (the method used in the SML/NJ compiler), a second that uses a two-continuations approach, and a third that uses a one-continuation-displacement approach [9]. 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. All measurement in the experiments are made by counting the number of instructions executed by the simulated programs.
The program for the experiment 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 (we later increase this value), calling function **f** and making computations. The result of the computation in function **f** never raises an exception so the handler **ovfl** is never evaluated. The program was translated by three different compilers: the SML/NJ compiler (old), our compiler with the one-continuation approach, and our compiler with the two-continuation approach

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
```

We showed in [9] that this program produces exception handling overhead. Figures 3, 4, and 5 illustrate the performance of the program on its four different versions: with an exception handler in function f, with no exception handler (for the old and new approach), and with an exception handler using the two continuation approach [9,10]. The graphs show 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.

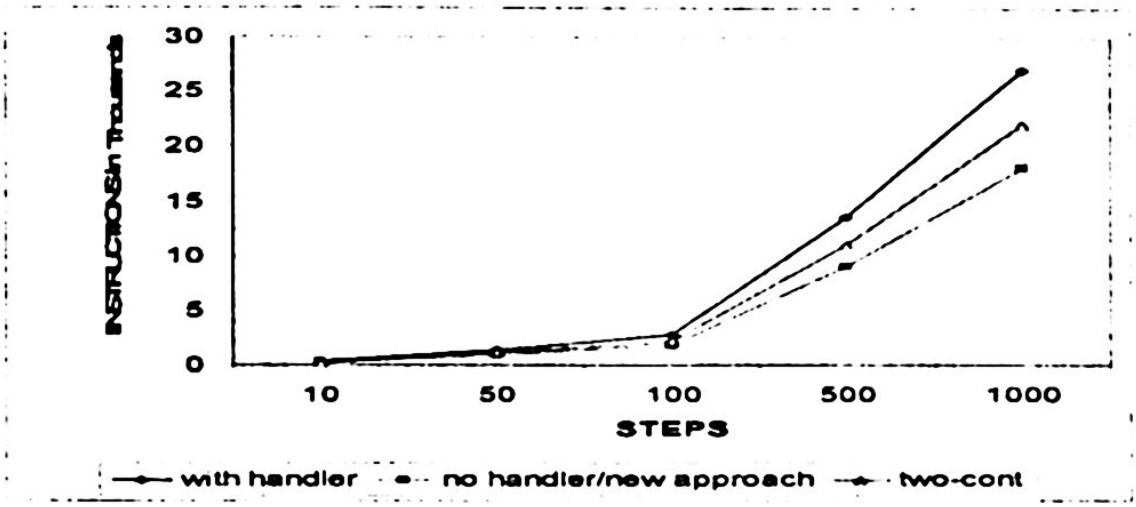

Fig. 3. Performance of program from 10 to 1000 steps

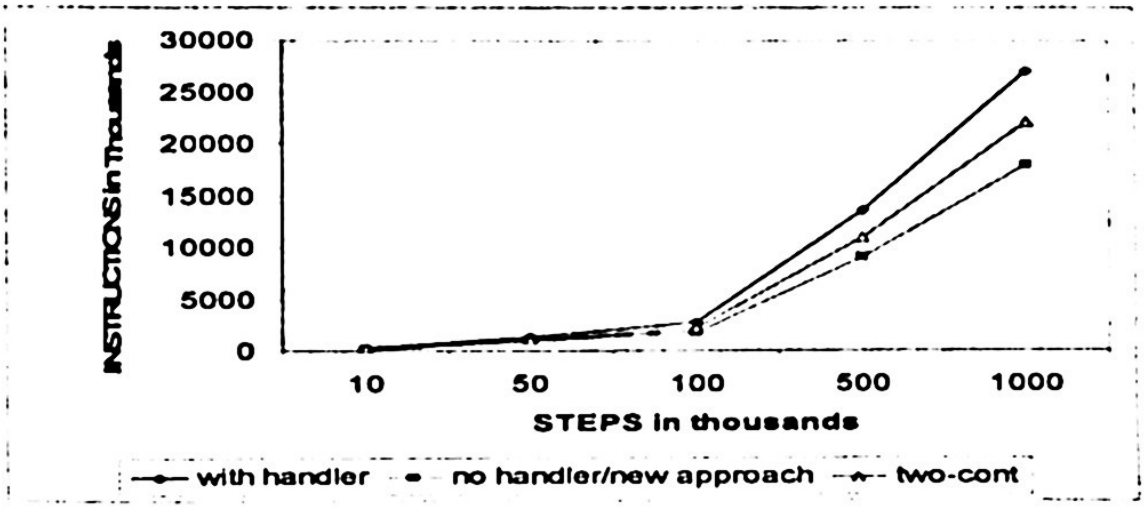

Fig. 4. Performance of program from 10000 to 1000000 steps

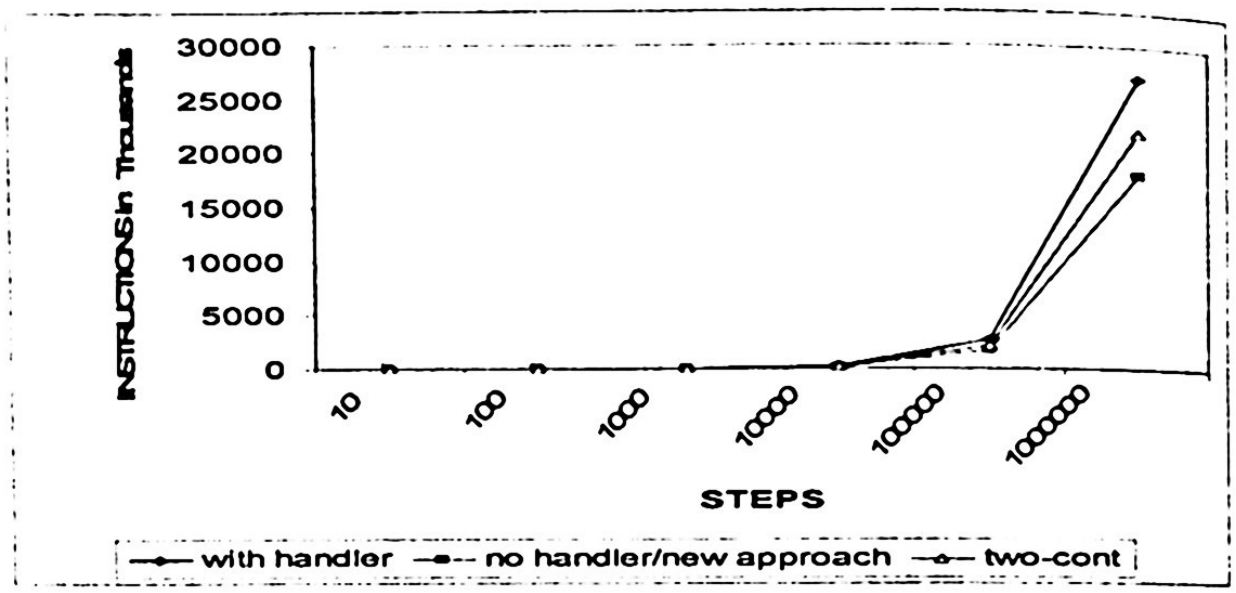

Fig. 5. Performance of program from 10 to 1000000 steps

4 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. Our model of translation and execution allows a programmer (or student/teacher) to write, translates, and executes programs in a source functional language (an extended lambda language) and a target abstract machine language. The model can be seen as a framework that can be used to execute programs, allowing studying a wide range of performance assessments. Last, we designed and implemented a new technique where the entire overhead is moved from the normal flow of control to the code executed when an exception is raised.

References

1. *American National Standard Programming Language PL/I.* ANSI X3.53-1976. American National Standards Institute, New Jork.
2. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, 1992.
3. Andrew Appel and David B. MacQueen. 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.
4. Baker, T.P. and Riccardi, G. A. Implementing Ada Exceptions. *IEEE software*, September 1986, 42-5.

5. Dinechin, C. de. C++ Exception Handling. *IEEE Concurrency*, Vol. 8, No.4, October-December 2000, Pages 72-79.
6. Venners, B. Inside the Java 2 Virtual Machine. McGraw-Hill, second edition, 1999.
7. Leroy (Xavier), Rémy (Didier), Vouillon (Jérôme) et Doligez (Damien). -- *The Objective Caml system release 2.04*. -- Rapport technique, INRIA, décembre 1999.
8. Milner, R., Tofte, M., and Harper Jr., R.W. *The Definition of Standard ML*. MIT Press, Cambridge, Massachussetts, 1990.
9. Ramon Zatarain. Design and Implementation of Exception Handling with Zero Overhead in Functional Languages. *PhD Dissertation*, Florida Institute of Technology, 2003.
10. Ramon Zatarain and Ryan Stansifer. Exception Handling for CPS Compilers. *Proceedings of the 41st ACM Southeast Regional Conference (ACMSE'03)*, Savannah, Georgia.
11. Reade, Ch. *Elements of Functional Languages*. Addison-Wesley, 1989.