Tail Call Optimization for Joos 1W [1]

Marianna Rapoport

April 8, 2013

[1] CS 644 Project Report
Instructor Ondřej Lhoták

This project aims to implement tail call optimization for *Juice*, a compiler from Joos[2] to Assembly. Tail call optimization allows writing deeply recursive functions without overflowing the stack.

[2] Joos (Java's Object Oriented Subset) is a subset of Java used for teaching and research.

Contents

Introduction

A *tail call* is a call site — a statement consisting of a function call — in tail position. A call site is said to be in *tail position* if it is the last statement that a function has to perform before returning.

An important type of tail call is tail recursion. A function is *tail recursive* if it invokes itself in a tail call.

Let's consider a Haskell function that calculates Fibonacci numbers:

```
1 fibo 0 = 1
2 fibo 1 = 1
3 fibo n = fibo (n - 1) + fibo (n - 2)
```

In line 3, `fibo` calls itself recursively twice. After the recursive call of `fibo(n - 1)` is performed, it has to be added to the result of `fibo(n - 2)`. The recursive call is not the last action performed before the result of `fibo n` is returned, and therefore this is not a case of tail recursion. Moreover, we have to recalculate `fibo(n - 2)` in the recursive call of `fibo(n - 1)` which makes `fibo` have exponential runtime.

To improve efficiency, we have to ensure that the evaluation of every previous member of the sequence is done only once. This can be done easily by using accumulators and tail recursion:

```haskell
fibo n = fibo2 0 1 n
  where fibo2 prev1 prev2 0 = prev2
        fibo2 prev1 prev2 n = fibo2 prev2 (prev1 + prev2) (n - 1)
```

Running this code will reveal that no stack overflow occurs even for large values of `n`: e.g. the result of `fibo 100000` will be calculated right away. Besides the linear complexity of the algorithm, the reason for the better performance is that the Haskell compiler performs *tail call optimization* (TCO). Because the return value of `fibo2` is the return value of the tail call, we can pop `fibo2`'s stack frame from the call stack, push the stack frame for the tail call, and replace the tail call's return address with `fibo2`'s return address. For a large sequence of nested tail calls, this optimization reduces the number of necessary stack frames from linear to constant.

Motivation

Unlike compilers of functional languages, the Java Virtual Machine (JVM) does not support TCO.

However, the implementation of many object-oriented design patterns in Java can result in stack overflows even when all methods use tail recursion[3].

For instance, according to good object-oriented programming practice, Class Hierarchies should be used to represent Unions[4]. Here is an example similar to what M. Felleisen, one of the creators of the Racket programming language, demonstrated on the European Conference on Object-Oriented Programming, 2004. To implement a list data structure, we create an abstract class `List<T>` that can either be `Empty` or a pair `Cons` of an element of type `T` and the rest of the list:

[3] Comment by Matthias Felleisen on J. Rose's, "Tail calls in the VM" article, blogs.oracle.com/jrose/entry/tail_calls_in_the_vm

[4] Bloch, J. "Effective Java." Prentice Hall, 2008.

```java
abstract class List<T> {
    int howMany() { return size(0); }
    abstract int size(int i);
}

class Empty extends List<T> {
    int size(int i) { return i; }
}

class Cons extends List<T> {
    T element;
    List<T> rest;
    int size(int i) { return rest.size(i + 1); }
}
```

If we run a test program

```java
class Test {
    boolean main(int n) {
        List<Integer> list = ... // create a list with n Cons's
        return list.howMany() == n;
    }
}
```

on `main(100000)`, we get a StackOverflowError which would not happen if Java supported tail call optimization.

It is difficult to implement TCO for Java because the JVM supports *stack inspection* which impairs program transformations like TCO[5].

The objective of this project is to implement TCO for Joos. Joos is a subset of Java that does not compile to the JVM and does not have the mentioned limitations that would hinder us from implementing TCO.

[5] Fournet, Cedric, Gordon. "Stack inspection: Theory and variants." ACM SIGPLAN Notices 37.1 (2002): 307-318.

Implementation

The implementation of TCO consists of two parts:

- finding out which statements are tail calls
- modifying code generation logic for method invocations to readjust the stack frame of the caller function

For every source Joos class, a list of tail call statements `tailCalls` is retrieved according to the first part. This is done during the static analysis of the program. The list is then passed to and stored in the `CodeGenerator` class. Before the code for a method invocation is generated, we check if that method invocation's node is contained in `tailCalls`. If it is, the method invocation code will be optimized as TCO. In the next two sections we describe the implementation of the two parts in detail.

Retrieving tail call statements

We consider a statement to be in tail position according to the following definition[6].

Definition 1 (Tail call)**.** A method invocation m is a tail call if it satisfies one of the following conditions:

1. It is the right-hand side of a `return` statement in a non-void method[7];
2. In a void method, m is the last statement of a statement s, and s is immediately followed by a `return` statement[8];
3. m is the last statement of a void method.

[6] For Java, this definition would be insufficient. Whenever a statement were surrounded by a `try/catch` block, the tail position would depend on the existence of a `finally` block.

[7] i.e. `return` m`;`

[8] e.g. m`; return;` (here, $s = m$)

Definition 2 (Last statement)**.** A statement s is said to be the last statement of another statement[9] b if s and b satisfy one of the following conditions:

[9] A statement can be either single or compound.

1. b is a single statement, and $s = b$.
2. b consists of a sequence of statements $s_1, \ldots, s_n$, and $s = s_n$.
3. b is a conditional statement: $b =$ `if(` c `) {` t `} else {` e `}`, where c is a conditional expression, t and e are statements, and s is the last statement of t or of e.

If b is a `while` or `for` loop, it does not have a last statement.

Determining if a statement is a tail call is done with two implementations of the visitor pattern[10]: one that gets the last statement of a method, and one that gets the statement before a `return` statement in void methods.

[10] Erich, Gamma, et al. "Design patterns: elements of reusable object-oriented software." Reading: Addison Wesley Publishing Company (1995)

Assume we got to a `return` statement in a void method, and we need to recursively get the last statement of the block that precedes the `return` statement. We will traverse the tree of that preceding block looking for all statements that could be executed last. However, if we were to traverse a block of statements looking only for statements that come before `return` statements, we would not include the statements that will be executed last in that block. This is why we need to traverse the tree in two different modes, using two visitors.

The first visitor, `TailCalls`, finds method invocations in `return` statements of non-void methods and invokes the second visitor, `LastStatements`, which finds the last statements of a statement. For every Abstract Syntax Tree (AST) node, both visitors return a list of statements.

For non-void methods, `TailCalls` returns the method invocations on the right-hand sides of `return` statements. For void methods, it does two things:

- finds all `return` statements and invokes `LastStatements` on the statement s preceding the `return` statement, if s is not a loop
- invokes `LastStatements` on the body of method declaration nodes

The list of statements returned by `LastStatements` is added to the list of `TailCalls` statements.

`LastStatement`, on the other hand, finds last statements according to definition 2.

The resulting list of tail call statements is passed into the `CodeGenerator` visitor pattern constructor.

Code Generation

The code generation for a method invocation of a method m has the following outline, if m is *not* in tail position:

```
; Push arguments a_1, ..., a_n
;; evaluate argument a_1, put result into eax
push eax          ; first argument
...
;; evaluate argument a_n, put result into eax
push eax          ; last argument

; If m is not static
;; evaluate implicit this, put result into eax
push eax          ; implicit this

call m_label      ; call method m

; If m is not static
add eax, 4        ; pop implicit this

add eax, n * 4    ; pop arguments
```

A typical stack frame for a non-static method is shown on figure 1. If the method is static, the stack frame will not include space for the implicit `this`.

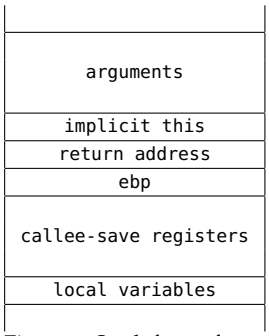

Figure 1: Stack frame for non-static method. In our case, callee-save registers (CSR) are `esi`, `edi`, and `ebx`

In what follows we will assume that both the caller and callee methods are non-static, i.e. need an implicit `this` to be pushed after their arguments. If a method is static, the arguments on the stack will be immediately followed by the return address of the method.

Consider a method M (the caller) that has a method invocation m in tail position.

The idea of TCO is to reuse the caller's stack frame to store the callee's one. In order to do this, instead of calling m and returning to M, we *jump* to m and return to M's return address after m has finished executing. However, since we won't return to M to pop the CSR, we will skip pushing them in m in order to pop them only once (we push in M, but pop in m). To skip saving CSR in m, instead of jumping to m's label, we jump to the position in m that is right after the CSR saving, using a special label for TCO invocations.

When we jump to the right position in the method declaration of m, we need to make sure the arguments for m's invocation are placed in the right position on the stack. To replace M's stack frame with m's, we

- replace the arguments and implicit `this` of M with the arguments and implicit `this` of m
- leave the return address, `ebp`, and CSR intact

Since the number of arguments for M and m might be different,

we cannot just replace the caller's agruments with the callee's ones. For example, if m has more arguments than M, data on the stack that precedes M's arguments will be overridden if we maintain the data that should not be moved on the same place.

To get around this problem, just before we jump to m's declaration, we first push all the contents of the beginning of m's stack frame into the stack [11]. Then we move the pushed values up the stack, so that m's first argument overrides M's first argument. After the readjustment of the stack, the stack pointer is moved to point to the new location of the last CSR.

[11] m's arguments, m's implicit 'this', M's caller's return address, M's caller's ebp, and M's CSR

Testing

As a form of regression testing, the TCO was enabled, and the *Juice* compiler was run against the Marmoset test suite; no additional test failures occured.

To actually test that the TCO implementation achieves its purpose, we ran tail-recursive functions with one argument for which the amount of tail call invocations was proportional to the value of its argument. For a large argument value, the function would caused a stack overflow (seen as a segmentation fault) in the TCO-disabled mode of the compiler. With TCO enabled, the function ran fine. Here is a simple example of a testing function:

```
1  public int f(int x) {
2     if (x > 123) return f(x - 1);
3     return 123;
4   }
5
6   public static int test() {
7     return new recurse().f(100000000);
8   }
```

Conclusions

Optimization of tail calls is a useful feature to have, because recursion is a powerful and expressive programming construct. TCO is usually associated with implementations of functional programming languages. However, by doing simple modifications to the mechanism for allocations of new stack frames, we can add it to a more traditional ALGOL-like imperative language.