Tail Call Optimization for Joos 1W [1]

Marianna Rapoport

March 10, 2013

The goal of this project is to implement tail call optimization for the Joos[2]-to-Assembler compiler *Juice* written in Java. Tail call optimization avoids stack overflows in functions involving tail recursion.

A *tail call* is a function call in tail position. An expression is in *tail position* when evaluating the expression is the last 'action' that a function has to perform before returning.

If a function calls itself in a tail call, the function is said to be *tail recursive*. Let's consider an example of a factorial function in Haskell:

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

In line 3, `fact` calls itself recursively; after the result of `fact(n - 1)` is obtained, it has to be multiplied by `n`. The recursive call is not the last action performed before the result of `fact n` is returned, and therefore this is not a case of tail recursion.

Every time `fact` calls itself recursively, a new stack frame is allocated on the call stack, which can cause a stack overflow for large `n`.

Let's rewrite the factorial function in a tail-recursive way:

```
fact n = fact2 1 n
    where fact2 acc 0 = acc
          fact2 acc n = fact2 (n * acc) (n - 1)
```

Running this code will reveal that no stack overflow occurs even for large values of `n`. What is happenning here is that the Haskell compiler is performing a *tail call optimization* (TCO). Because the return value of `fact2` is the return value of the tail call, we can pop `fact2`'s stack frame from the call stack, push the stack frame for the tail call, and replace the tail call's return address with `fact2`'s return address. For a large chain of tail calls, this optimization saves a significant amount of memory on the stack.

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

However, as argued by Matthias Felleisen[3], "a language should implement TCO in support of proper design"[4]. The reason is that the implementation of many object-oriented design patterns in Java will likely result in stack overflows even when all methods use tail recursion: "Java isn't TCO [...], meaning it doesn't allow programmers to design according to OO principles."

[1] CS 644 Project Proposal

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

[3] One of the creators of the Racket programming language

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

As an example, let’s look at the object-oriented principle of using Class Hierarchies for Unions[5] that Felleisen demonstrated at the 18th ECOOP[6]. 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:

```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. The same test in a Haskell program as follows runs just fine:

```haskell
data List a = Empty | Cons a (List a)

howMany :: List a -> Int
howMany list = size 0 list
    where size n []     = n
          size n (x:xs) = size (n + 1) xs

main :: Int -> Bool
main n = let list = ... -- create list of size n
         in howMany list == n
```

The problem with implementing TCO in Java is that the Java Virtual Machine supports *stack inspection* which invalidates program transformations like TCO[7].

Given the more simple nature of Joos, implementing TCO should be possible, at least for certain scenarios. This will be the objective of my project.

[5] Bloch, J. “Effective Java.” Prentice Hall, 2008.

[6] European Conference on Object-Oriented Programming, 2004, Oslo

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