Tail-call optimization in R_5

Caner Derici and Ryan Scott

May 4, 2016

(*Nota bene*: The code for this project is located at `https://github.iu.edu/cderici/P523/tree/tail-call-optimization`. Our changes were regression-tested against all of the original tests for R_1 through R_7 in the `tests/` subdirectory, and we also added new tests, prefixed with `tco_`, to test our new additions.)

Abstract

Tail-call elimination is an important optimization that functional languages heavily rely upon. It is based on the observation that a stack frame for a function call in a tail position can be eliminated by reusing the caller's frame. This makes tail recursion computationally equivalent to looping constructs (e.g. `for`, `while`) in imperative languages.

We implemented tail-call elimination (TCO) on R_5 by making functions jump to the callee in the assembly level instead of making regular function calls, resulting in manual handling of the stack space. This allowed us to be able to reuse the same stack space for function calls, which essentially kept the space complexity at a constant factor. We also had to change the closure representation, which allows us to handle higher-order functions as well.

1 Tail calls

A tail call is a function call in a tail position inside of a function. A tail position is a fixed syntactic region of code that signifies the idea of a "last action of a function". For example, consider the following structurally recursive factorial function.

```
(define (fact n)
  (cond
    [(<= n 1) 1]
    [else (* n (fact (sub1 n)))]))
```

The `(fact (sub1 n))` call inside of this function is *not* in a tail position. For example, when `(fact 5)` is invoked, the result of the recursive call `(fact (sub1 5))` has to be multiplied with `5`, i.e. there's still something to do after the function call. Because of this, we cannot overwrite the stack frame of `(fact 5)`, since we still have to keep the `5` until after `(fact (sub1 5))` returns (to multiply with it).

However, consider the following tail-recursive factorial function.

```
  (define (fact-tail n acc)
    (cond
      [(<= n 1) acc]
      [else (fact-tail (sub1 n) (* n acc))]))
```

Now the only action that a `(fact 5 1)` call will perform with the result of `(fact 4 5)` is to return it. The key observation here is that the results of `(fact 5 1)` and `(fact 4 5)` (and all the subsequent calls in the computation sequence) are the same. Figure 1 shows the stack progression along with the return values for each function invocation.

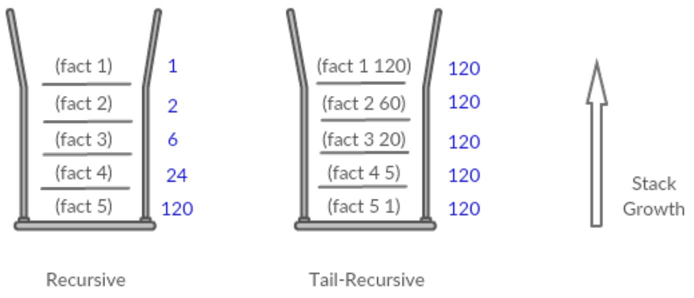

Figure 1: Stack Progressions of Regular and Tail Recursive Calls

This observation is the central idea behind tail-call elimination, as the two subsequent tail calls can use the same stack frame. In a recursive function, this essentially keeps overwriting the stack frame of the first call, making the whole computation safe-for-space.

2 Function calls in R5

R_5, as originally defined in its reference implementation, doesn't take into account whether function calls are in tail position, so it is completely oblivious to any possibility of reusing the stack for tail calls. An unfortunate consequence of this design is that it is quite easy to define recursive R_5 functions that use up all available stack space. Real-life examples of such functions include deeply tail-recursive functions and infinite loops, e.g.,

```
(define (explosion [n : Integer]) : Integer
  (explosion (+ n 1)))

(explosion 0)
```

Without TCO, the generated assembly code for this function will essentially be an infinite sequence of `callq`s and `subq`s on the stack pointer, resulting in an explosion of stack frames (hence the name). This is a problem. Not only does this R_5 not meet the requirements of the Scheme report (which requires implementations to be tail-recursive), but we can't even make infinite loops that are *infinite*!

3 Project goals

When we first proposed this project, we divided up the work into three milestones, in order of increasing difficulty:

1. Apply tail call optimization only to first-order functions which use immediate tail recursion.
2. Apply tail cail optimization to first-order functions that are mutually tail-recursive.
3. Apply tail cail optimization to all tail cails, even those involving higher-order function arguments and lambdas.

We were confident in our ability to complete 1 and 2, but not as certain whether we could successfully complete 3. In our original estimate, we believed that implementing 3 correctly would involve applying inter-procedural static analysis to determine which code paths one could go through in a program. However, we realized that achieving 3 was much simpler if we adopted a strategy of storing multiple labels in a closure (which we detail in the following section).

This approach turned out to work extremely well in R_5, and required none of the fancy static analysis that we were afraid to implement. As a result, we successfully completed all three milestones.

4 Code changes

Alongside several minor fixes, we achieved TCO in R_5 by implementing two major changes:

- Create the necessary entry labels for functions to be invoked (i.e. jumped) from a tail position, and producing unconditional `jmp`s (instead of regular `callq`s) for all the tail calls inside.
- Changing the internal closure representation to contain the tail-call entry label alongside the regular label carried by the `function-ref`.

4.1 Labeling

Before, R_5 produced a single assembly label for each function, which is used when making function calls. We created an additional label called the *entry label*, which is always suffixed with `Entry`. An entry label is used to enter the function when it's invoked (i.e. jumped) from a tail position inside of its caller function. Below is the code produced for the `explosion` function with both kinds of labels:

```
        .globl explosion
explosion:
        pushq   %rbp
        movq    %rsp, %rbp
```

```
        subq    $32, %rsp
        movq    %r14, -8(%rbp)
        movq    %r13, -16(%rbp)
        movq    %r12, -24(%rbp)
        movq    %rbx, -32(%rbp)
explosionEntry:  <========

        movq    %rdi, %rbx
        movq    %rsi, %r13
        movq    %rdx, %r13
        leaq    explosionEntry(%rip), %r14
        movq    free_ptr(%rip), %r8
        addq    $16, %r8
        cmpq    fromspace_end(%rip), %r8
        setl    %al
        movzbq  %al, %r8
        cmpq    $0, %r8
        je      then20251
        jmp     end20252
then20251:
        movq    %rbx, %r8
        addq    $0, %r8
        movq    %r8, %rdi
        movq    $16, %rsi
        callq   collect
end20252:
        movq    free_ptr(%rip), %r8
        addq    $16, free_ptr(%rip)
        movq    $3, 0(%r8)
        movq    %r14, 8(%r8)
        movq    $0, %r14
        addq    $0, %r13
        movq    8(%r8), %r12
        movq    %rbx, %rdi
        movq    %r8, %rsi
        movq    %r13, %rdx
=====>  jmp     *%r12

        movq    -8(%rbp), %r14
        movq    -16(%rbp), %r13
        movq    -24(%rbp), %r12
        movq    -32(%rbp), %rbx
        addq    $32, %rsp
        popq    %rbp
retq
```

Having a label *after* the function prelude in the assembly allows us to eliminate the regular stack handling when entering the function. Of course, this requires adjusting the stack (e.g. putting the function arguments in the argument registers and stack arguments) right before the jump at the call site. Additionally, all the tail calls inside of the function are no longer `callq`s, but rather unconditional `jmp`s.

4.2 Closure representation

One of the crucial changes in implementing tail-call elimination in R_5 involves the closure representation. Originally, closures were represented by a vector containing a function reference and the free variables inside of the function:

$$(lambda : (ps \ldots) : rt\ body) \mapsto (vector\ label\ fvs \ldots)$$

The problem with this representation is especially apparent when dealing with higher-order functions. Since higher-order functions can be passed around as parameters, there is no way to know from the function reference itself if the function is going to be used in tail position or not (i.e. it's not clear which label to use to call/jump to). Therefore, we add to the closure representation the entry label, along with the regular label for the function:

$$(lambda : (ps \ldots) : rt\ body) \mapsto (vector\ label\ entry\text{-}label\ fvs \ldots)$$

This allows us to retrieve the entry-label from the closure when a function is used in tail-position, and the regular label otherwise. This way, we can always be assured that whenever we invoke `callq` or `jmp` in assembly, we are using the correct label.

4.3 Changes in the passes

Along with the changes to labeling and closure representation, a couple of passes needed to be improved to handle the new forms and implement the necessary functionality.

- **reveal-functions**: This is the part of the compiler that marks whether function calls are in tail position. In order to do this, we added an additional parameter called `tail?` to indicate at every step whether we are in a tail position or not. When revealing a function call, we produce a `tail-app` node to indicate we're making a tail call if `tail?` is true, and produce an `app` node otherwise.
- **convert-to-closures**: We changed the handling for `function-ref`s and lambdas to produce closure code using the new representation.
- **flatten**: When flattening the program, we treat `tail-app`s as simple expressions and don't assign them to a variable. This is to reflect the idea that a tail call is the last operation that the function will perform.
- **select-instructions**: We handled the `tail-app` node very similarly to how we handle the the `app` node. We first put the rootstack variable into `%rdi`, move the arguments into the argument registers, and then perform an unconditional jump (`jmp`) in place of a `callq`.
- **print-x86-64**: We changed the code that generates the preludes and conclusions such that all functions reserve the exact same amount of stack space in their preludes, and take back an equal amount in their conclusions. This ensures that if a function tail-calls into another, that the appropriate amount of space will be added back to the stack pointer. If not, the caller function's stack might get clobbered!

5 Measuring the benefits of TCO

To instill a sense of appreciation for how much time and space TCO saves when running compiled programs, we developed two microbenchmarks designed to demonstrate the benefits of TCO. The first such microbenchmark is the Ackermann function A, which is defined mathematically as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

A is of particular interest to computer scientists not only because of its interesting computability properties, but also because it can lead to extremely deep levels of recursion, making it useful as a performance benchmark. Note that while we can define A straightforwardly in R_5, it doesn't show off the benefits of TCO as much as it could due to the call to A in non-tail position in the third case. Therefore, the program we will benchmark will be converted to continuation-passing style to leverage more tail calls:

```
(define (ackermann-cps [cont : (Integer -> Integer)]
                       [m : Integer] [n : Integer]) : Integer
  (if (eq? m 0) (cont (+ n 1))
      (if (eq? n 0)
          (ackermann-cps cont (+ m (- 1)) 1)
          (ackermann-cps
            (lambda: ([x : Integer]) : Integer (ackermann-cps cont (+ m (- 1)) x))
            m (+ n (- 1))))))
(define (ackermann [m : Integer] [n : Integer]) : Integer
  (ackermann-cps (lambda: ([x : Integer]) : Integer x) m n))
(ackermann 3 5)
```

We compiled this program with two versions of our R_5 compiler— one with TCO enabled, and one without it—and ran the compiled code using the `/usr/bin/time` program on Linux.

The results for the compiler with TCO:

```
$ /usr/bin/time -v ./a.out
253     ...
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
        ...
        Maximum resident set size (kbytes): 3252
        ...
        Minor (reclaiming a frame) page faults: 525
        ...
```

The results for the compiler without TCO:

```
$ /usr/bin/time -v ./a.out
253     ...
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
        ...
        Maximum resident set size (kbytes): 8560
        ...
        Minor (reclaiming a frame) page faults: 1848
        ...
```

The difference is clear: enabling TCO saves about 5.3 MB worth of RAM usage, and prevents 1323 page faults! From a memory perspective, TCO is an obvious win.

What is suprising, however, is that both programs run nearly instantly. We would have expected the code with TCO to run a little slower, but in practice, this does not turn out to be the case. In face, even with the following microbenchmark designed to churn through millions of function calls:

```
(define (make-your-cpu-get-hot [x : Integer] [y : Integer]) : Integer
  (if (eq? 0 x)
      42
      (make-your-cpu-get-hot (+ x (- y)) y)))
(let ([a-little-warmer (lambda: () : Integer (make-your-cpu-get-hot 100000 1))])
  (let ([a0   (a-little-warmer)])
  ...
  (let ([a499 (a-little-warmer)])
    (a-little-warmer))...))
```

Both compilers, with and without TCO, produce code which finishes in less than one second. We learned that adjusting the stack pointer is not as computationally expensive as we had originally thought. Still, from the perspective that fewer x86 instructions = faster runtime, TCO is certainly preferable, and from a memory usage perspective, there is demonstrable evidence that TCO wins.

6 Design considerations and future work

We wish to emphasize that while our work successfully adds TCO to R_5, it is by no means an *optimal* solution. In particular, we made an extreme simplification by requiring that all top-level `define`s allocate (and deallocate) the exact same amount of stack space. Although this makes it possible for a series of tail calls to successfully finish in any `define`d function, it is also quite wasteful, since all functions now must allocate as much stack space as possible.

It is difficult to envision a robust solution to this problem, since given the existence of higher-order functions, it can be difficult to know statically all possible paths through `define`d functions. One idea is to change the generated code such that closures also store how much stack space the underlying function requires, and creating a different conclusion for tail calls that `addq`s `%rsp` an amount equal to the stack space number inside the closure. There would have to be an additional runtime cost associated with checking if a function should `jmp` to the normal conclusion or the special tail-call conclusion, but the cost may be worthwhile given the amount of memory this technique could save.

Another area for improvement is in the generated code for closures themselves. Currently, closures have two separate entries for non-tail-call labels (e.g., `foo`) and tail-call label (e.g., `fooEntry`). This seems like a code smell, since we always know that a tail-call label is just the original label with the suffix `Entry`. It may be possible, using some assembly trickery, to retrieve the non-tail-call label and concatenate `Entry` on the end to synthesize the tail-call label. If such a thing is possible, it would allow us to save 8 bytes per closure.

Space considerations notwithstanding, our project proves that TCO is inside the realm of possibility for R_5, and it can successfully coexist with more advanced language features like higher-order function arguments and lambdas. This is good news, because it means that even first-time compiler authors can successfully create a Scheme that adheres to the specification!