Tail-Call Optimization

Principles of Programming Languages

Colorado School of Mines

`https://lambda.mines.edu`

Motivation

1. Recursion is a beautiful way to express many algorithms, as it typically makes our algorithm easier to prove.
2. Calling a function requires space on the call stack for the variables, parameters, and return address from the call.
3. What if we could translate *certain kinds* of recursion into loops at compile time so that we could use it feasibly?

Tail-Calls

A **tail-call** is a function call for which the return value of the call becomes the return value of the function. For example, `sqrt` is the tail-call of this function:

```c
double distance(struct point a, struct point b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

A function which is **tail-recursive** only ever calls itself as a tail call.

Racket Example

Consider this Racket function:

```
(define (sqrt x guess)
  (if (< (abs (- (expt guess 2) x)) 0.001)
    guess
    (sqrt x (* 0.5 (+ guess (/ x guess))))))
```

- The only place `sqrt` ever calls itself is in a tail position
- `if` is a **macro**: it returns something to be evaluated rather than evaluating it itself
- Since `(sqrt x (* ...))` is one of the returned forms from `if`, it is a tail call.

Practice 1

Identify the tail-calls and state whether the function is tail-recursive or not.

```
(define (double-all lst)
  (if (null? lst)
    '()
    (cons (* 2 (car lst))
          (double-all (cdr lst)))))
```

Practice 2

Identify the tail-calls and state whether the function is tail-recursive or not.

```
(define (sum-list lst)
  (if (null? lst)
    0
    (+ (car lst)
       (sum (cdr lst)))))
```

Practice 3

Identify the tail-calls and state whether the function is tail-recursive or not.

```scheme
(define (sum-list lst)
  (define (sum-iter lst acc)
    (if (null? lst)
      acc
      (sum-iter (cdr lst) (+ (car lst) acc))))
  (sum-iter lst 0))
```

Practice 4

Identify the tail-calls and state whether the function is tail-recursive or not.

```
(define (fib n)
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

Practice 5

Identify the tail-calls and state whether the function is tail-recursive or not.

```
(define (fib n)
  (define (fib-iter n a b)
    (if (= n 0)
      a
      (fib-iter (- n 1) b (+ a b))))
  (fib-iter n 0 1))
```

Translating Tail-Recursion to Loops

To translate a tail recursive function to a loop, we can surround the function body in an infinite loop, and when we tail-call ourselves, replace that with a reassignment of our arguments and a `continue`.

How might this look in the code below?

```
int fib_iter(int n, int a, int b) {
    if (n == 0) return n;
    return fib_iter(n - 1, a, b);
}
```

Loop Translation Example

Before optimization:

```
int fib_iter(int n, int a, int b) {
    if (n == 0) return n;
    return fib_iter(n - 1, a, b);
}
```

After optimization: (not pretty, but shows what a compiler might do)

```
int fib_iter(int n, int a, int b) {
    while (true) {
        if (n == 0) return n;
        int next_n = n - 1, next_a = b, next_b = a + b;
        n = next_n, a = next_a, b = next_b;
        continue;
    }
}
```

Can we do better?

What if we wanted our compiler to handle circular recursive cases like this:

1. Procedure `A` can tail-call procedure `B`.
2. Procedure `B` can tail-call procedure `C`.
3. Procedure `C` can tail-call procedure `A`.

Brainstorm optimization techniques with your learning group. Think aloud!

The Trampoline Method

Idea: in interpreted languages, we have an *evaluator* function which takes an expression and the storage context, if we make this function call functions in a loop and have our functions return what it should call next, we can get free tail-call optimization that also supports circular tail calls!

(figure on whiteboard)

Thinking of the Trampoline Visually

Figure: Think of these people as functions, one function returns its unevaluated tail-call to the trampoine rather than calling it itself.

Why is TCO important to SlytherLisp?

The only structure for looping in SlytherLisp is recursion, so we need to be able to efficiently be able to implement loops in SlytherLisp.

You should implement TCO using the trampoline method. **Hint:** `lisp_eval` can be your trampoline.

Tail Recursive Modulo Cons

Some tail call optimizers have an additional trick in their hat: **tail call modulo cons.** This technique allows the CDR argument to a cons tail call to become a tail call.

1. Rather than waiting for the CDR to finish evaluating to construct the cons cell, the cell is constructed right away without setting the CDR argument.
2. A reference to the CDR is passed back to the trampoline.
3. The trampoline stores the result of the CDR evaluation into the CDR reference.

The Racket compiler implements this technique. **You are not required to implement this in SlytherLisp.**