Proper Tail Calls

CPSC 509: Programming Language Principles

Ronald Garcia*

7 November 2015

A peculiar phenomenon

Consider the following program written in the C language:

```c
int doh(int x) {
  return doh(0);
}

int main() {
  return doh(0);
}
```

It defines a procedure `doh` that calls itself. Well that's not very interesting: it looks like it will just call itself forever.[1] What happens if we compile and run it? Let's see:

```
> cc doh.c
> ./a.out
Segmentation fault: 11
```

Well, that failed miserably! Even more peculiar though, is what happens if we compile it with optimizations turned on and run it again.[2]

```
> cc -O3 doh.c
> ./a.out
>
```

Now that's interesting: the program finished immediately! So let me get this straight: without optimizations the program crashes in an epic ball of flames: with optimizations turned on an infinite loop finishes instantly. Now that's what I call a powerful optimization!

For comparison, though, consider the corresponding Racket code:

```racket
(define (doh x)
  (doh 0))

(doh 0)
```

If you run this program in Racket (or any other decent Scheme implementation), it will run forever.[3]

What gives! Well, the first lesson of the day is that turning on optimization in a typical C compiler (I used clang-500.2.75) can actually change the observable behaviour of a program: C is a peculiar language that way. But neither of those programs ran forever as we might have guessed.

We get a bit more insight if we play the same game in Java:

*© Ronald Garcia.

[1]Those of you who know C can probably guess what's going to *really* happen!

[2]C programmers may not see this one coming!

[3]The longest I've run it before forcing it to stop is a couple of days, but I saw no reason for it to stop.

```
public class Doh {

    public Doh() { }
    public static void main(String args[]) {
        doh(0);
    }

    public static int doh(int x) {
        return doh(0);
    }
}
```

Compiling and running this program tells us something useful, albeit not very satisfying:

```
> javac Doh.java
> java Doh
Exception in thread "main" java.lang.StackOverflowError
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        at Doh.doh(Doh.java:11)
        ...
        ...
```

Sadly it still doesn't run forever, but at least you get a stack trace (a long one)! We see that Java fails with a *stack overflow*.

You might be scratching your head now: why doesn't Java run forever? Why doesn't Racket fail with a stack overflow? Below we explain what's going on here using semantics as our guide.

1 Procedures, Recursion, and "call stacks"

For this study, we consider a small variant of TFL with procedures and recursion.

$$\begin{array}{l}
x \in \textsc{Var}, \quad n \in \mathbb{Z}, \quad t \in \textsc{Term}, \quad v \in \textsc{Value}, \quad E \in \textsc{ECtxt}, \\
l \in \textsc{Loc}, \quad \mathrm{SO} = \textsc{Value}, \quad \sigma \in \textsc{Store} = \textsc{Loc} \xrightarrow{\text{fin}} \mathrm{SO}, \quad \textsc{Pgm} = \{\, t \in \textsc{Term} \mid FV(t) = \emptyset \,\}
\end{array}$$

$$\begin{array}{rcl}
t & ::= & x \mid n \mid t\ t \mid \lambda x.t \mid \mathsf{rec}\ x.t \\
v & ::= & n \mid \lambda x.t \mid l \\
E & ::= & \square \mid E[\square\ t] \mid E[v\ t]
\end{array}$$

$$\boxed{\longrightarrow \subseteq (\textsc{Pgm} \times \textsc{Store}) \times (\textsc{Pgm} \times \textsc{Store})}$$

$$\begin{array}{rcll}
E[(\lambda x.t)\ v], \sigma & \longrightarrow & E[[l/x]t], \sigma[l \mapsto v] & l \notin dom(\sigma) \\
E[\mathsf{let}\ x = v\ \mathsf{in}\ t], \sigma & \longrightarrow & E[[l/x]t], \sigma[l \mapsto v] & l \notin dom(\sigma) \\
E[\mathsf{rec}\ x.t], \sigma & \leadsto & E[[(\mathsf{rec}\ x.t)/x]t], \sigma &
\end{array}$$

The semantics above is a reduction semantics with state. We don't bother wrapping the program and store in angle-brackets. That's just a stylistic thing.

This semantics has a twist. Just like the language of mutable variables, we have a store and we allocate a new store location for each variable, but we do not support mutation of locations. Why in the world would we do that? Well, in a certain sense, this is a more realistic language implementation model when we think about comparing to a language like Java, and it will help us talk about important relevant issues.

First let me explain the realism aspect. A language like Java typically has complex values like objects. But objects are not typically allocated directly on the stack: they exist in a part of memory called the heap, which we model using a store. This way, the language doesn't have to literally copy all of the contents of an object every time you want to share it with other code.[4] Typically simple machine integers are not store-allocated, but we'll store-allocate them anyway just to get across the ideas in this section as simply as possible. Imagine that we're using Java `Integer` objects.

The language above gives us enough machinery to model the example program from the last section.

$$(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{x}.\mathsf{doh}\ \mathsf{x})\ 0$$

We use rec to define a recursive function in-place (we could bind the whole thing to a variable and then use it, but that's just cosmetics) and immediately apply it to 0.

Let's examine how this reduces, with annotations for the evaluation context:

$$\begin{array}{lll}
 & t & \sigma \\
 & \underline{(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{x}.\mathsf{doh}\ \mathsf{x})\ 0} & ,\emptyset \\
\longrightarrow & \underline{(\lambda \mathsf{x}.(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{y}.\mathsf{doh}\ \mathsf{y})\ \mathsf{x})\ 0} & ,\emptyset \\
\longrightarrow & \underline{(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{x}.\mathsf{doh}\ \mathsf{x})}\ \mathsf{l}_0 & ,[\mathsf{l}_0 \mapsto 0] \\
\longrightarrow & (\lambda \mathsf{x}.(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{y}.\mathsf{doh}\ \mathsf{y})\ \mathsf{x})\ \underline{\mathsf{l}_0} & ,[\mathsf{l}_0 \mapsto 0] \\
\longrightarrow & \underline{(\lambda \mathsf{x}.(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{y}.\mathsf{doh}\ \mathsf{y})\ \mathsf{x})\ 0} & ,[\mathsf{l}_0 \mapsto 0] \\
\longrightarrow & \underline{(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{y}.\mathsf{doh}\ \mathsf{y})}\ \mathsf{l}_1 & ,[\mathsf{l}_0 \mapsto 0, \mathsf{l}_1 \mapsto 0] \\
\longrightarrow & (\lambda \mathsf{y}.(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{z}.\mathsf{doh}\ \mathsf{z})\ \mathsf{y})\ \underline{\mathsf{l}_1} & ,[\mathsf{l}_0 \mapsto 0, \mathsf{l}_1 \mapsto 0] \\
\longrightarrow & \underline{(\lambda \mathsf{y}.(\mathsf{rec}\ \mathsf{doh}.\lambda \mathsf{z}.\mathsf{doh}\ \mathsf{z})\ \mathsf{y})\ 0} & ,[\mathsf{l}_0 \mapsto 0, \mathsf{l}_1 \mapsto 0] \\
\longrightarrow & \vdots & \vdots
\end{array}$$

For clarity, I am alpha-renaming some bound variables so that you can tell more easily which variable reference matches which binding. The redex in each case is underlined, meaning that anything not underlined is part of the evaluation context.

Now looking at this reduction sequence, we can make a few observations. First, if you ignore the difference in location that appears in the code, the sequence is going to keep repeating over and over again. Along the way, we see only three kinds of evaluation context surrounding a redex: the empty context $\square$, application with a location in argument position $\square[\square\ l_n]$, and application with our big lambda in operator position $\square[(\lambda.\ldots)\ \square]$. The context doesn't grow past a single frame, and really the context corresponds exactly to that concept that no one who teaches programming can seem to avoid: *the stack*. The stack is a representation of all the work left to do in your program.

One thing to observe, though, is that our store σ grows by one entry with each function call, so it sure looks like it will grow without bound. Could this be the source of our stack overflow in Java? Well, *no*. Java is a *garbage-collected* language, which means that occasionally it checks to see if there are any *unreachable* store locations, meaning that the location doesn't appear in the program nor in the binding of a store location. If we assume a very aggressive garbage collector that runs on each program step, then our

[4]For simplicity, we're not modeling cloning semantics at all here.

reduction sequence turns into the following:

$$
\begin{array}{lll}
 & t & \sigma \\
 & \underline{(\mathsf{rec\ doh}.\lambda \mathsf{x.doh\ x})}\ \mathsf{0} & ,\emptyset \\
\longrightarrow & \underline{(\lambda \mathsf{x}.(\mathsf{rec\ doh}.\lambda \mathsf{y.doh\ y})\ \mathsf{x})\ \mathsf{0}} & ,\emptyset \\
\longrightarrow & \underline{(\mathsf{rec\ doh}.\lambda \mathsf{x.doh\ x})}\ \mathsf{l}_0 & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & (\lambda \mathsf{x}.(\mathsf{rec\ doh}.\lambda \mathsf{y.doh\ y})\ \mathsf{x})\ \underline{\mathsf{l}_0} & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & \underline{(\lambda \mathsf{x}.(\mathsf{rec\ doh}.\lambda \mathsf{y.doh\ y})\ \mathsf{x})\ \mathsf{0}} & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & \underline{(\mathsf{rec\ doh}.\lambda \mathsf{y.doh\ y})}\ \mathsf{l}_1 & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & (\lambda \mathsf{y}.(\mathsf{rec\ doh}.\lambda \mathsf{z.doh\ z})\ \mathsf{y})\ \underline{\mathsf{l}_1} & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & \underline{(\lambda \mathsf{y}.(\mathsf{rec\ doh}.\lambda \mathsf{z.doh\ z})\ \mathsf{y})\ \mathsf{0}} & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & \vdots & \vdots
\end{array}
$$

The moment that the program dereferences l_0, it becomes inaccessible and so is collected. Thus the store ends up having only one binding at a time for the duration of the computation.

So what's up?

2 Java is broken!

Essentially, The answer is that Java is using more space than is strictly necessary, as is C. The semantics we defined above essentially corresponds to what Racket does (including the store allocated variable bindings!). One of the key steps is the reduction rule for function application, which simply substitutes a location for a variable in the body of the function. That action is sure to not increase the size of the stack. In fact, looking at the evaluation contexts, the only time the stack is extended is when evaluating an operator or evaluating an operand. *Function calls have nothing to do with it!*

On the other hand, Java and C extend the call stack *on every function call*. We can model this behaviour by modifying our language as follows:

$$
\begin{gathered}
t ::= \ldots \mid \lambda.\mathsf{return}\ t \mid \mathsf{return}\ t \\
E ::= \ldots \mid E[\mathsf{return}\ \square] \\
E[\mathsf{return}\ v], \sigma \longrightarrow E[v], \sigma
\end{gathered}
$$

Now a new feature, return, has been added that does nothing but return its value. Furthermore, function definitions have return baked right into their syntax. As a result, every function call introduces a return that must be dealt with. Notice that evaluating inside the body of return introduces a context (i.e., stack) frame! With these changes, keeping aggressive garbage collection, we get a new reduction sequence:

$$
\begin{array}{lll}
 & t & \sigma \\
 & \underline{(\mathsf{rec\ doh}.\lambda \mathsf{x.return\ (doh\ x)})}\ \mathsf{0} & ,\emptyset \\
\longrightarrow & \underline{(\lambda \mathsf{x.return}\ ((\mathsf{rec\ doh}.\lambda \mathsf{y.return\ (doh\ y)})\ \mathsf{x}))\ \mathsf{0}} & ,\emptyset \\
\longrightarrow & \mathsf{return}\ (\underline{(\mathsf{rec\ doh}.\lambda \mathsf{x.return\ (doh\ x)})}\ \mathsf{l}_0) & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & \mathsf{return}\ ((\lambda \mathsf{x.return}\ (\mathsf{rec\ doh}.\lambda \mathsf{y.return\ (doh\ y)})\ \mathsf{x})\ \underline{\mathsf{l}_0}) & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & \mathsf{return}\ (\underline{(\lambda \mathsf{x.return}\ (\mathsf{rec\ doh}.\lambda \mathsf{y.return\ (doh\ y)})\ \mathsf{x})\ \mathsf{0}}) & ,[\mathsf{l}_0 \mapsto \mathsf{0}] \\
\longrightarrow & \mathsf{return}\ (\mathsf{return}\ (\underline{(\mathsf{rec\ doh}.\lambda \mathsf{y.return\ (doh\ y)})}\ \mathsf{l}_1)) & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & \mathsf{return}\ (\mathsf{return}\ ((\lambda \mathsf{y.return}\ ((\mathsf{rec\ doh}.\lambda \mathsf{z.return\ (doh\ z)})\ \mathsf{y}))\ \underline{\mathsf{l}_1})) & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & \mathsf{return}\ (\mathsf{return}\ (\underline{(\lambda \mathsf{y.return}\ ((\mathsf{rec\ doh}.\lambda \mathsf{z.return\ (doh\ z)})\ \mathsf{y}))\ \mathsf{0}})) & ,[\mathsf{l}_1 \mapsto \mathsf{0}] \\
\longrightarrow & \vdots & \vdots
\end{array}
$$

As you can see, this program is accumulating returns, so it is bound to use up more and more space as it evaluates. As such it's inevitable that the Java and C programs will crash.

The fact that Racket supports procedure calls that do not accumulate any space is called *proper implementation of tail calls*. Some places in the literature call this *tail call optimization*, but that name is just patently broken, for a few reasons. First off, as per our semantic model, *function calls never cause the stack to grow*: it's other expressions like complex expressions in operator or operand position that cause the stack to grow. Functions are viewed as the cuplrit because recursive (or mutually recursive) function calls are the only way that we can witness the growing stack in response to expressions that have evaluation contexts like let or application.

Second, proper tail calls are not an optimization. To put it bluntly, it's just strange to call the absence of stupid behaviour an "optimization." [5]

3 C has crazy semantics

But what about this thing with the optimized C program ending immediately? Well, this is an artifact of how C was designed, to support optimizations that might make a program run faster, but don't guarantee that behaviour will stay the same. Support for these behaviour-changing optimizations is part of why C can be optimized to go so fast: some assumptions are made about how programs work, and they can lead to very unexpected and hard to find bugs! In particular, any "looping" code that performs no visible effects in C is deemed by the standard to be equivalent to a no-op. If you'd like to see a semanticists perspective on the trash fire that is C, I highly recommend Robbert Krebbers' PhD. Thesis Krebbers [2015].

4 Why do languages do this?!?

Now, let's set aside the polemics and talk about why programming languages work this way. One of the main reasons is because the most popular programming languages over time have had this behaviour. Back in the dark ages, getting recursion to work *at all* was a miraculous thing. FORTRAN originally couldn't handle recursion, but LISP did, but without proper tail calls. Furthermore, machine architectures implemented procedure calls as single instructions that would push a return address and jump to a new location: even assembly language did not implement proper tail calls! So it was natural to use these features to implement languages like C and friends. It was only with the development of Scheme in the 1970s that the notion of proper tail calls was discovered [Steele, 1977]. So later languages inherited the same procedure call semantics as earlier languages.

Another reason is related to how programmers are used to debuggers working. In general, we assume that a debugger can give us a call trace whenever an error occurs, but in order to do that, the runtime system must record what function calls were made. This has led dynamic languages like Python to build this into the language runtime semantics.

Finally, since many imperative languages have built-in looping constructs, which do not accumulate stack space, programmers are expected to use those instead of function calls to implement loops. Have you ever noticed that the typical debugger doesn't record how many iterations through a loop you have gone through before an error occurs? This is a situation where debuggers compromise on what error information they capture, deciding that function calls are more important than loop iterations. More recently, folks have been developing *omniscient* debuggers, notably Mozilla's `rr` debugger,[6] uses record-and-replay technology to capture execution and support backwards winding. Naturally, the seed behind this idea was planted by Bil Lewis's, a long-time Schemer, who understood that the call stack is a useful but insufficient abstraction for debugging.[7]

On the other hand, Racket programmers are quite used to using recursion to implement loops, and languages like Racket provide *loop macros* that are implemented under the hood in terms of function calls.

[5]e.g., "Mac OSX implements the 'running-Safari-doesn't-erase-my-hard-drive' optimization."
[6]https://rr-project.org/
[7]https://www.youtube.com/watch?v=7VE93LLYw54

The steppers that we have written in this class are implemented that way (using the *named let* notation for defining and calling recursive functions).

Finally, some languages forego proper tail calls in order to support *memory safety* without requiring garbage collection. For example, the Rust programming language, initiated and primarily developed by folks at Mozilla, automatically manages your program memory in a stack-like fashion, even if values were allocated on the heap. This lets Rust run on embedded devices without requiring a lot of memory or a complex runtime system that demands a garbage collector. The cost of this, though, is that every program variable is tied to a marker on the function call stack called a *region* [Grossman et al., 2002]. Regions are collected in a stack-like fashion, tied to function calls. In the case of Rust, sacrificing proper tail calls was a conscious decision in the interest of developing a low-level programming language. C, on the other hand, isn't memory safe anyway, so accumulating stack space at every function calls doesn't buy it much.

References

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512563. URL `http://doi.acm.org/10.1145/512529.512563`.

R. Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, December 2015. URL `https://robbertkrebbers.nl/thesis.html`.

G. L. Steele. Debunking the "expensive procedure call"" myth or, procedure call implementations considered harmful or, lamdba: The ultimate goto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. URL `http://dspace.mit.edu/handle/1721.1/5753`.