Even Lower Order Functions for Returning
or
Why would anyone ever want to use a -6 order function?

With apologies to Chris Okasaki

Saul Field*, Dann Toliver, Erik Derohanian

March 2023

Abstract

Higher order functions are well studied in the CS literature, and provide affordances in functional programming ranging from map and reduce to parser combinators[1]. Lower order functions have received considerably less attention. We present a framework for understanding lower order functions, and show how they may be used to encapsulate many imperative concepts in a functional setting. In particular, the operations of delete, return, and the stack smashing of continuations are put into a functional framing.

1 Introduction

There is a longstanding history, dating back to Turing and Church, of the imperative and functional camps wrestling with each other[1]. Each side desires to subsume the other, to absorb its twin and gain its strength while retaining its own form.

Lower order functions are a winning strategy in this age old struggle. Unlike higher order functions, which increase the possibility space with each successive level, building abstractions on top of abstractions, lowering order functions actually simplify your process each time they are invoked.

This simplifying aspect maps directly to a variety of imperative features that are otherwise complicated to express in the functional paradigm. Delete, return, and even exit, that bugbear of the imperative world, can be expressed naturally and functionally using lower order functions.

*Author declares the existence of a competing interest, having received financial contributions from Big Integer

[1]So heated were these matches that they eventually lead to the complete separation of Church and state.

2 First-order functions

First-order functions do not accept functions as arguments or return functions. They accept or return base values (non-function primitives), and may have other side effects.

These environment semantics for a first-order function in a typical functional programming language clearly reveal the state encapsulation and immutable semantics typically associated with languages derived from the lambda calculus.

$$\mathit{def}\ \frac{}{E \vdash \texttt{fun}\ x \to e \Downarrow (\texttt{closure}\ x \to e, E)}$$

$$\mathit{apply}\ \frac{E \vdash e_1 \Downarrow (\texttt{closure}\ x_c \to e_c, E_c) \quad E \vdash e_2 \Downarrow v_2 \quad (E_c, x_c \mapsto v_2) \vdash e_c \Downarrow v}{E \vdash e_1\ e_2 \Downarrow v}$$

$$\mathit{eval}\ \frac{E \vdash f \Downarrow (\texttt{closure}\ x \to e, E)}{\mathit{eval}(E, f, x) = v \dashv E'}$$

Some well-known, straightforward first-order functions are the Ackermann function [2], the Collatz conjecture [3], Fast Inverse Square Root [4], and the Standard Model Lagrangian [5]. These functions are simple and easy to reason about, and will rarely pose any operational difficulty for software developers.

First-order functions are typically considered the base case for higher order functions: a sixth order function is a function that takes a function that takes a function that takes a function that takes a function that takes a function that takes a base primitive as its argument as its argument as its argument as its argument as its argument as its argument.

However, there is no particular reason to bottom out with first-order functions. Let us continue this degression into the underworld, beginning with the end, the null point of infinity, the first and greatest among numbers – zero.

3 Noop as a zero-order function

Zero order functions take no arguments and provide no return value, yet are terribly useful. When a higher order function absolutely positively has to take a useless function as an argument, noop has no substitutes.

Many imperative languages have statements like "debugger", which is effectively a noop from the perspective of the computational process, but is effectful with respect to the external environment. By treating statements like these as zeroth order functions we effectively gain the effects of algebraic effects on their effects, capturing the imperative effectfulness in a snuggley functional embrace.

3.1 Environment semantics for zero order functions

$$\mathit{eval}\ \frac{E \vdash f \Downarrow (\texttt{closure}\ x \to e, E)}{\mathit{eval}(E, f, x) = v \dashv E}$$

4 Minus-first-order functions

Like zero-order functions, a minus-first-order function neither takes arguments nor returns values. It does, however, have an effect: it removes a variable from the environment.

In particular, it removes the last variable that was bound into the environment, so it is no longer available. When using de Bruijn indices it reduces the value stack by one, like "drop" in Forth.

This allows the modeling of imperative statements like "delete" in a purely function setting. It also creates a path for adding linear logic or other variable ownership constraints to a programming language – any language with first-class minus-first-order functions and zero-class function definitions allows construction of novel memory management models directly in user space.

4.1 Environment semantics for minus-(first,second,third)-order functions

We introduce a new symbol here, '●' (pronounced *VOID*) to denote the degree of destruction of the resulting environment.

$$eval^{-n}\frac{E \vdash f^{-n} \Downarrow (\texttt{closure}\ x \to e, E)}{eval(E, f^{-n}, x) = v \dashv E^{\bullet^n}}$$

5 Minus-second-order functions

The next step beyond dropping a single value from the environment is to drop an entire frame. This is the functional equivalent of the imperative "return" function, but now reified as a first-class value by minus-second-order functions.

Of course, the return function can already be emulated by using continuations in languages derived from the untyped lambda calculus, but this is a bit like using a wrecking ball to drive a finish nail. In conjunction with minus-third-order functions, the ability to precisely target individual stack frames provides a much more nuanced story around control flow management and theory.

6 Minus-third-order functions

Continuations are an extremely useful control flow construct, but they have also long been a source of confusion and frustrating bugs, and have not seen widespread mainstream adoption outside a few fringe use cases.

A minus-third-order function continues the work of a minus-second-order function, but consumes all the stack frames instead of just one per evaluation. This allows the invocation of a continuation to be modelled as a pair of functions, one which consumes all the current stack frames, and a second which adds frames that were captured earlier.

This division of labour allows more precise control, and enables uses that would otherwise be difficult to achieve: if you have captured a full continuation, but want deploy it without entirely consuming your current stack, this separation allows the fine grained control needed. (The alternative case, where a delimited continuation is captured but we desire to invoke it as a full continuation, is considered next.)

7 Minus-fourth-order functions

Early termination is very common in imperative programming models, where a system may halt and return a result or error message prematurely upon some condition, regardless of how deeply nested its current state is. However, this is quite difficult to simulate in the lambda calculus and similar systems, and is usually appended as a statement the underlying runtime supports with no proper theoretical framing.

A minus-fourth-order function takes minus-third-order functions one step further, by not only eliminating all the stack frames but the actual process as well. This not only provides the necessary construction for incorporating a sudden exit into the theory, it also lifts it into a proper first class function that can be treated as a normal value in the system.

7.1 Environment semantics for minus-fourth-order functions

Note here that there is no longer any environment to speak of after this evaluation, and of course the return value is also *VOID*.

$$eval^{-4}\frac{E \vdash f^{-4} \Downarrow (\texttt{closure}\ x \to e, E)}{eval(E, f^{-4}, x) = \bullet \dashv \bullet}$$

8 Minus-fifth-order functions

A minus-fifth-order function not only removes the process that invokes it, but also eliminates the possibility of being invoked in the future, leading to a radical simplification of the process and perhaps even the computational substrate – operating system, firmware, and perhaps the broader network.

Various systems in the past have eliminated erroneous code as a safety measure[6], but a true minus-fifth-order function is considerably involved to construct, as it must not only remove all copies of its own code, but also remove the possibility of itself being run in the future.

An practical example of a minus-fifth-order function is a kind of security construction, where a flaw is exploited to remove the flaw. For instance, a network virus that utilizes a zero day exploit in order to patch every machine it compromises, destroying itself in the process.

8.1 Environment semantics for minus-fifth-order functions

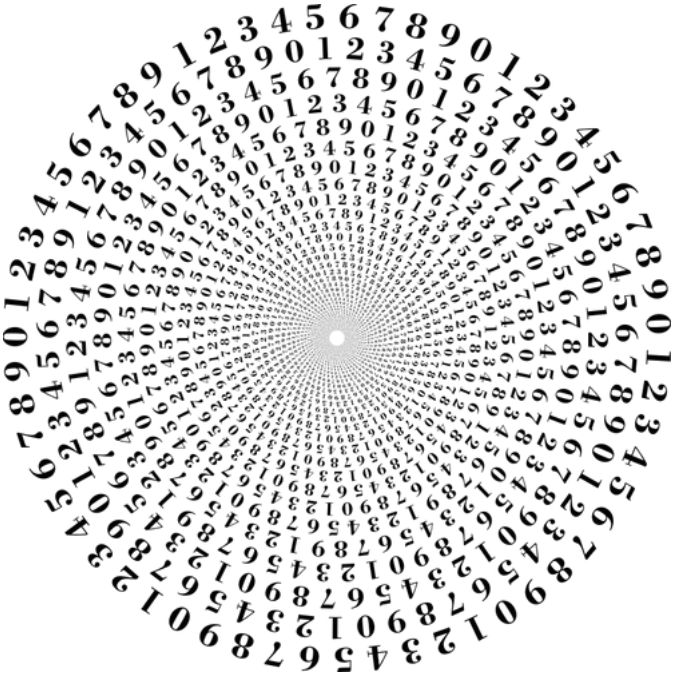
The semantics here are identical to the minus fourth order semantics, with a notable difference. Not only have the value and environment succumbed to the *VOID*, but the *eval* construct of our metalanguage has as well.

$$eval^{-5}\frac{E \vdash f^{-5} \Downarrow (\texttt{closure}\ x \to e, E)}{eval(E, f^{-5}, x) = \bullet \dashv \bullet[eval \rightsquigarrow \bullet]}$$

9 A minus-sixth-order function

Still lower? There is a minus-sixth-order function – the function which prevents anyone who discovers it from talking about it, or even thinking about it, by eradicating the neural pathways capable of evaluating its construction during its evaluation. Or it certainly seems that there should be such a function. The authors, who have designed effective biotemporal computation signalling systems and who believe they have been certified in the field of antimemetics, appear to have spent the last twelve years designing it with the intent to include it in this paper, but it seems to have escaped them.

9.1 Environment semantics for minus-sixth-order functions



References

[1] C. OKASAKI, "Even higher-order functions for parsing or why would anyone ever want to use a sixth-order function?," *Journal of Functional Programming*, vol. 8, no. 2, p. 195–199, 1998.

[2] W. Ackermann, "Zum hilbertschen aufbau der reellen zahlen," *Mathematische Annalen*, vol. 99, pp. 118–133, Dec. 1928.

[3] Wikipedia, "Collatz conjecture — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Collatz_conjecture`, 2023. [Online; accessed 21-March-2023].

[4] K. N. William Kahan, "e_sqrt.c." `http://www.netlib.org/fdlibm/e_sqrt.c`, May 1986.

[5] G. Lisi, "The standard model lagrangian." `http://deferentialgeometry.org/papers/Standard%20Model.pdf`, 2007.

[6] B. Nystrom, "Vigil." `https://github.com/munificent/vigil`, 2013.