Making the Theoretical Practical: The Untyped λ-Calculus in C++

Jim McCann*
TCHOW llc

Abstract

While of great theoretical importance, the untyped λ-calculus is not often used outside of a handful of toy examples.

Imagine our excitement when we learned that C++11 "supported lambdas"; finally, we thought, all the features of a **production** language – standard libraries, memory support, in-line assembly, syscalls, and strong typing – paired with the wonderful programming style afforded by the untyped lambda calculus. The possibilities seemed endless.*

Alas, our hopes were dashed when we learned that by "lambdas", what was actually meant were strongly typed anonymous functions which had reasonable capture semantics and generally meshed nicely into the flow of C++ programs, replacing cumbersome function objects and obtuse macros. In this paper we show how to snatch victory from the ashes of defeat by using C++ lambdas to implement *real*, untyped lambda calculus lambdas and binding. This implementation is based on an in-place execution model for the lambda calculus that uses an *argument stack*.

CR Categories: _.__. [Blank]: Blankings—Blank

1 Introduction

The λ-calculus [**?**] is a model of computation that proceeds through substitution ("β-reduction").

$$x[x := r] \to r \tag{1}$$
$$(\lambda x.y)z \to y[x := z] \tag{2}$$

While it may be straightforward to implement this in terms of copy-pasting of program fragments, self-modifying code can often result in performance penalties owing to instruction cache flushes. Therefore, we instead introduce an execution semantics for the λ-calculus involving a stack and instruction pointer.

Our execution model is similar to the standard reduction, in that our instruction pointer will move through unbound variables within the λ-calculus term in the same order that the traditional substitution-based semantics would have after substitution.

And when we say "our", we're probably neglecting some background work but it's late-o'-clock so.

Y'know.

Sorry about that.

2 Background

The λ-calculus was introduced by Turing [**?**], and – famously – proved to be equivalent to the much more practical model of computation proposed by Jacquard [**?**]. It has enjoyed an important place in compuphilosophical cannon since, with the exception of a period in the mid-1700's it was briefly suppressed by the breakaway *Mechanists*, Lovelace and Babbage.

We shall neglect to mention von Neumann, nor the entirety of the nineteenth through twentieth centuries in this discussion. Suffice it to say, LISP was a bad idea [**?**].

In the present day, reliable complex systems [**?**; **?**] are engineered in reasonable and entirely satisfactory languages like C++ and its poor imitator Scala [**?**].

3 Execution Model

While the conventional "execution" of lambda term involves β-reduction – that is, finding applications and bound variables and performing substitutions – our execution model is more straightforward, involving two stacks and an instruction pointer.

We first describe the basic operations in terms of this model and then formalize the semantics:

At the beginning of a lambda term, the instruction pointer is set to the first character of the term, the argument stack is empty and the return address stack is empty.

`L`*`X`*`.`*`body`*

A λ term. When encountered, pops the top value from the argument stack and stores it in a location (memory or register) which will be referred to as *X* within *body*. These addresses can be statically allocated in a pre-pass of the term since they are purely based on position within the uninterpreted term.

*e-mail: ix@tchow.com
*Or, at least, roughly stack-depth.

```
std::vector< std::function< void() > > arguments;
#define APPLY_BEGIN   ([&]( std::function< void() > fn_){ arguments.emplace_back(fn_);
#define APPLY_MIDDLE   })([&](){
#define APPLY_END   });
#define LAMBDA( X )   auto X = arguments.back(); arguments.pop_back();
```

Figure 1: *Encoding of untyped lambda calculus into typed lambda expressions in C++11.*

When *X* is encountered in *body*, the address of the next instruction is pushed onto the return address stack and then the instruction pointer is moved to *X*.

(*func*:*arg*)

An application. At `(`, pushes the address of the `:` onto the argument stack. (That is, makes *arg* available for any inner lambda term.) At the end of *func*, execution jumps to after `)`. (That is, *arg* isn't executed inline.) Finally, when *arg* is being executed, just before *)*, the return address is popped and set to the instruction pointer.

3.1 Formal Definition

As is typical in programming languages papers, we formally define the execution model by providing x86-assembly-like definitions for each character:

; code for `L`
```
POP X
```

; code for when *X* appears in the body:
```
CALL X
```

; code for `(`
```
PUSH :label
```

; code for `:`
```
JMP :end
:label
```

; code for `)`
```
RET
:end
```

3.2 In C++

Our implementation of the untyped lambda calculus in C++ (Figure 1) consists of four macros and a global arguments stack. Essentially, the `APPLY_` macros capture the address of code by using two C++11 lambdas and passing the second as an argument to the first, which pushes it onto the argument stack. Each lambda uses automatic reference capture semantics, which allows it to communicate with variables from the outside scope, but also establishes in inner scope (which conveniently limits the lifetime of variables defined by `LAMBDA`).

And, best of all, it actually seems to work.

For example, it finally allows one to easily count to four:

```
int32_t count = 0;
APPLY_BEGIN
    APPLY_BEGIN
       LAMBDA( X )
       LAMBDA( Y )
       APPLY_BEGIN
          Y();
       APPLY_MIDDLE
          APPLY_BEGIN
             Y();
          APPLY_MIDDLE
             X();
          APPLY_END
       APPLY_END
    APPLY_MIDDLE
       cout << ++count << endl;
    APPLY_END
APPLY_MIDDLE
    LAMBDA( A )
    A();
    A();
APPLY_END
```

Which is equivalent to:

((λ X. λ Y. (Y (Y : X)) : cout << ++count << endl) : λ A. AA)

Which I think we can all admit is a lot less cumbersome than a for loop.

3.3 Safety

Intel hasn't made computers with auto-destruct instructions for years, and other manufacturers are taking their cue. We predict code produced in this way will be safe by 2014.

Of course, arguments are only removed from the argument stack by lambdas, so things might get really weird if you apply something without a lambda to something else. But that's the joy of the untyped lambda calculus. And, besides, optimizers thrive on undefined behavior.

```
APPLY_BEGIN
    LAMBDA( A )
    APPLY_BEGIN
       A();
    APPLY_MIDDLE
       A();
    APPLY_END
APPLY_MIDDLE
    LAMBDA( Y )
    APPLY_BEGIN
       Y();
    APPLY_MIDDLE
       Y();
    APPLY_END
APPLY_END
```

Figure 2: *Yes, this compiles and does what you think it should, though not for very long (though* `ulimit` *can help).*

4 Limitations

This section intentionally left blank.

5 Bibliography

This section unintentionally left blank.