CS525 Term Paper

Ezra Newman

December 2023

1 Lambda Calculus

Lambda calculus is a model for the theory of computation developed by Alonzo Church. It's formally equivalent to a Turing machine, but has some nice properties that make it easier to reason about in some situations.

Lambda calculus is based around the idea of *substitution* based computation, as opposed to a Turing machine's *state manipulation* based computation. This is related to the distinction between functional and imperative programming in higher level languages.

1.1 Grammar And Basic Rules

Pure lambda calculus describes only a tiny number of language constructs and operations, but those are sufficient to be a full computation model. Here is the complete BNF grammar:

$$\begin{array}{lcl} \langle\ \lambda\ \mathit{expr}\rangle & ::= & \langle \mathit{var}\rangle \\ & | & \lambda\ \langle \mathit{var}\rangle\ .\ \langle\lambda\ \mathit{expr}\rangle \\ & | & \langle\lambda\ \mathit{expr}\rangle\ \langle\lambda\ \mathit{expr}\rangle \end{array}$$

There are two objects here– λ expressions and variables. λ expressions are "first-class" constructs; they are values and are given as inputs to functions; substitute in for variables, etc.

Here's a concrete example of a simple expression in lambda calculus, the identity:

$$\lambda x.x$$

This expression is composed of two parts; the part before the dot describes the parameter (there must be exactly one) and the part after is the body, which can be a variable or another λ expression.

This is a value as much as it is a function, because functions are fist class citizens. If we apply something to it, it returns the input. For example:

$$(\lambda x.x)(y) \to y$$

During evaluation the lambda expression on the right $(\lambda x.x)$ is applied to the argument on the left (y). It consumes its argument and returns its body with all instances of x replaced by y.

1.1.1 Multi-parameter Functions

This grammar doesn't allow naive multi-parameter functions, but conveniently it's pretty trivial to construct them. I've included parentheses in this example to make grouping clearer, but they're not necessary:

$$\lambda x.\,(\lambda y.\,(x(y)))$$

Essentially, this is just a function that takes one argument, x, and returns another function that takes one argument, y. We curry x so that we can return $x(y)$.

This expression has the type $(A \to Z) \to A \to Z$.

1.2 Church Numerals and Booleans

Church numerals are a way of representing integers in pure lambda calculus. As we saw when looking at the grammar, there are no concrete values in lambda calculus (ie no `ints`, `bools`, etc). We can however represent them with the tools we do have.

Church numerals represent integers using the repeated application of a function. For example, 3 would be represented as:

$$\lambda f.\lambda x.f(f(f(x)))$$

And similarly 0 is just $\lambda f.\lambda x.x$.

1.2.1 Operators on Church Numerals

Pure lambda calculus is a complete language, so you can implement any computable operation using it. I will however limit this discussion to a few simple operations: increment, add, multiply, raise (exponentiation), decrement, and subtract.

1.2.1.1 Increment is the simplest, we just need to wrap our number in one more layer of function applications:

$$\lambda a.\lambda f.\lambda x.f(afx)$$

When we populate a with a Church numeral and perform the substitution, we find we've wrapped x in one more application of f then before. For example with $a = 0$, ie just the literal value $\lambda f.\lambda x.x$ (I'll use $\lambda f'.\lambda x'.x'$ to avoid confusion:

$$(\lambda a.\lambda f.\lambda x.f(afx))(\lambda f'.\lambda x'.x')$$
$$\lambda f.\lambda x.f((\lambda f'.\lambda x'.x')fx)$$
$$\lambda f.\lambda x.f((\lambda x'.x')x)$$
$$\lambda f.\lambda x.f(x)$$

We've applied successor to zero and gotten one, as expected. One can see how this would work for any natural number: if instead of x' we'd had $f'(x')$ (for example), we'd have manipulated that symbol in the same way, and it would have ended up wrapped in f just like x' was. So we'd have incremented 1 to 2.

1.2.1.2 Add is easy once we have increment, because of how we're representing numbers. Numbers are repeated function application, so we can use that fact to repeatedly apply the increment function. We substitute increment in for f and the first number in for x, and we've got addition. Here's the expression for $1 + 2$:

$$(\lambda f.\lambda x.f(x))(\lambda a.\lambda f.\lambda x.f(afx))(\lambda f'\lambda x'.f(f(x)))$$

Simplifying this expression is omitted for space, but it will evaluate to $\lambda f.\lambda x.f(f(f(x)))$, ie 3.

1.2.1.3 Multiply is easy for the same reason– we have a convenient tool for repeatedly performing a function, and multiplication is just repeated addition. This process is much the same, except we substitute in our add function from before for our multiply function. An example of simplifying the expression is omitted for space.

$$\lambda m.\lambda n.\lambda f.\lambda x.m(nf)x$$

1.2.1.4 Raise is the easiest yet. To raise a church numeral to a power, simply apply the base to the power. For example, m^n is just nm.

1.2.1.5 Decrement works by keeping track of a pair of numbers, a and b. We start a at 0 and b at 1, and repeatedly increment them both, so that a is always less than b. We can do this without building a loop, because our numbers apply a function repeatedly. In other words, they *are* loops. Then if we take the first number out of that pair, we're at one less than we started with.

$$\lambda m.\lambda f.\lambda x.n(\lambda g.\lambda h.(gf))(\lambda u.x)(\lambda u.u)$$

1.2.1.6 Subtract works the same way as add, but with decrement instead of increment.

This is very inefficient, but lambda calculus only guarantees computability, not efficient computability.

1.2.2 Boolean Logic and Conditionals

Church booleans are defined as follows:

True:

$$\lambda a.\lambda b.a$$

False:

$$\lambda a.\lambda b.b$$

Think of these as functions that take two arguments and throw away one, returning the other.

This has the nice property that 0 and False are the same value:

$$\lambda a.\lambda b.b \leftrightarrow \lambda f.\lambda x.x$$

1.2.2.1 If-else expressions are pretty simple to implement. If you have some expression that evaluates to a boolean, you just apply the then branch as it's first argument and the else branch as it's second. I.E.:

$$((test)(then)else)$$

We can try substituting in True and False for *test*. Here's True:

$$((\lambda a.\lambda b.a)(then)else)$$

$$((\lambda b.then)else)$$

$$then$$

And here's False:

$$((\lambda a.\lambda b.b)(then)else)$$

$$((\lambda b.b)else)$$

$$else$$

Clearly this is an if-else statement that returns the *then* branch if *test* is true, and the *else* branch if *test* is false.

1.3 Recursion and the Y-Combinator

Lambda calculus is a complete computation model, so it allows recursion and looping. In the section on Church numerals, we've shown an example of using loops that repeat a finite number of times. Here's the simplest example of an infinite loop:

$$(\lambda x.xx)(\lambda x.xx)$$

If we apply the function to the argument, we'll find ourselves back where we started. If we are able to attach some useful work to this loop (and an exit condition), we'll have working model of a while loop or recursion. That idea leads us to the Y-Combinator:

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Here you can see the same recursive structure as before.

The Y-Combinator is useful because in pure untyped lambda calculus we can't access a function from inside its body by using its name, so instead we pass in the same function as an argument, and we can make calls to that. It's easy to see how this allows us to implement recursion without accessing the name of the function from inside itself.

1.4 Moving Beyond Pure Lambda Calculus

Pure lambda calculus is formally very powerful, but it's a low level language and can be difficult to actually use.

We'll implement a few impure language constructs– primitives for integers and mathematical operations on them, mostly– to make working with it easier.

2 A Closure-Based Interpreter

A closure-based interpreter fundamentally operates under a similar principle to an imperative language's stack. At each level, we have a stack of scopes, and whenever we need to resolve a variable to a value we search downward until we discover a binding for that variable.

Here's an example in Python-like pseudocode:

```
def bar():
  # here, x = 5
  x = 10
  # here, x = 10

def foo():
   x = 5
   # here, x = 5
   bar()
   # here, x = 5
```

This simple enough for trivial examples like this, but what about this case?

```
def bar():
  # here, x = 5
  y = 7
  print(x+y)

def foo():
   x = 5
   # here, x = 5
   return bar

(foo())()
```

This should print `12`, because the value of `x` is curried when we return `bar` from `foo`.

That means that `foo` isn't just returning a set of instructions, it's returning a closure– a set of instructions *along with* an environment to execute them in.

A closure-based interpreter is fairly easy, conceptually. We treat lambdas as (instruction, context) tuples. The context is a list of name:value bindings. In other words, the context's type is list[tup[name, value]]. Whenever we apply an argument to a lambda, we extend the context with the name-value binding for the lambda's argument. For example, if we had:

$$(\lambda x. < body >)(10)$$

And our context was:

$$[y : 3, x : 5]$$

We'd execute *body* with this context:

$$[x : 10, y : 3, x : 5]$$

When resolving x to a value inside the lambda, we iterate through our list and take the first value of x we find. This way we always take the most recent (and therefore tightest scoped) value of x that we can. So in this case, inside $< body >$ we would resolve x to 10 and y to 3, as expected.

`TMfix` and `TMlet` operate in much the same way. `TMfix` is just `TMlam`, except we also extend the context with a binding for the lambda itself. `TMlet` is equivalent to `TMlam` immediately executed, with the new variable binding:

$(\lambda x. < body >)(< new\_value\_for\_x >) \leftrightarrow$ `let x = <new_value_for_x> in <body> end`

So we can do the same thing: evaluate the argument, extend the context with the new argument-name:argument-value binding, and evaluate the body.

The new context is valid only inside the body of the lambda. We essentially extend a copy of the context for the new lambda, and go back to using the old context outside the lambda.[1]

[1] We don't actually have to copy the context in memory because we're using a prepend-only linked list, so when the lambda finishes executing the caller resumes with the same context, except it's head points to the head of c before the lambda prepended anything.

3 Type Checking and Type Inference

3.1 Extending Lambda Calculus With Types

Pure lambda calculus is untyped. Typed lambda calculus is an extension of Lambda calculus with a simple type system added to it.

Because we have a few "primitive" variables and operators, we can start with their types:

Entity	Type
`TMnil`	nil
`TMint`	int
`TMbtf`	boolean (btf)
`TMchr`	chr
`TMstr`	str
`TMlam`	$X \rightarrow Y$, where X is the type of the argument and Y is the type of the body
`TMadd`, `TMsub`, `TMmul`, `TMdiv`, `TMmod`	(int, int) → int
`TMlt`, `TMlte`, `TMgt`, `TMgte`, `TMneq`, `TMeq`	(int, int) → btf
`TMstr_eq`	(str, str) → btf
`TMnot`	btf → btf
`TMand`, `TMor`	(btf, btf) → btf
`TMstr_len`	str → int
`TMstr_get_at`	(str, int) → chr
`TMstr_set_at`	(str, int, chr) → str
`TMprint`	int → nil
`TMprchr`	chr → nil
`TMprstr`	str → nil
`TMref_get`	reference<X> → X
`TMref_set`	(reference<X>, X) → nil
`TMref_new`	X → reference<X>
`TMlist_new`	list<X>, where X is an unconstrained type variable
`TMlist_cons`	(X, list<X>) → X
`TMlist_nilq`, `TMlist_consq`	list<any> → btf
`TMlist_uncons1`	list<X> → X
`TMlist_uncons2`	list<X> → list<X>
llist operators have the same types, except with llists instead of lists.	
`TMif`	(btf, X, X) → X
`TMfst`	tup<X, Y> → X
`TMfst`	tup<X, Y> → Y
`TMtup`	(X, Y) → tup<X, Y>

Most of these are self explanatory. A few notes on the one's that aren't:

Some functions take in more than one argument. This is an extension of pure lambda calculus, but it's just syntactic sugar. There's two ways to implement this in pure lambda calculus: either with a closure over the arguments, eg a term of type $X \rightarrow Y \rightarrow Z$. That's a function with one argument that returns another function that takes one argument, which returns something of type Z. (Because the value of the first function's argument is curried into the second, it's available there). This has some nice advantages: for one, you can create a function `timesFive` of type *int* → *int* like this: (*multiply*)(5). The 5 is closed over in the return.

The other way is to have a function take in a tuple of all of it's arguments, which it then can break down internally. This doesn't allow you to create `timesFive` in the same way by partially populating arguments because all the arguments must be populated together.[2] This is how the closure-based interpreter works for these extremely low-level arithmetic, logic, and string operators. For user-space functions, either is possible but I've preferred the former when practical.

There are also a few operators that deserve special mention: TMlet, TMvar, and TMapp. TMlet has an identity type; it's type is whatever the input's type is. However, it has a side effect: it adds an additional constraint on it's variable input. For example, if the value input was constrained to *int*, then the variable assigned to that value inherits those constraints. TMvar checks for these constraints.

TMapp is basically the same, except it has the type $(X \to Y), X) \to Y$. In other words, it checks the type constraints of the lambda it's given.

3.2 Enforcing Type Constraints At Compile Time

Because we have type constraints for all our abstract syntax tree objects, we can traverse the syntax tree and solve for those constraints. As we traverse the tree of terms, we recursively process each term's children, maintaining a set of constraints as we do. At each object, we attempt to solve the constraints for that object given the context we've built up until this point. Here's a simple example:

```
TMlet("x", TMstr("not an int!"), TMadd(TMint(10), TMvar"x"))
```

We'd process `TMlet`. To do so, we'd process the `TMstr`, which is terminal; we return `TMstr`. We bind `x`'s type to *str* in our environment, and then process the body, the `TMadd` expression. We type check it's first argument, `TMint(10)`, and confirm that it is an *int*. Next we process the second argument, `TMvar"x"`. We check `x` against our context and find that it's a *str*. This conflicts with `TMadd`'s requirement that it be an *int*, so we throw an error.

If `x` *had* been an *int*, the `TMadd` would have finished type checking and returned *int*. `TMlet` would return that result, and so our final expression would have type checked to *int*, which is correct.

3.3 Generics

This implementation doesn't support generics. [3] Because we're dealing with ASTs only and not concrete syntax, there's a simple enough work around: if we have a generic function, we can just repeat the expression again for each type instead of binding and reusing it. Because we're building ASTs by hand in a fully featured host language (ATS), this is really easy; we can manipulate the syntax tree however we'd like before handing it to the type checker and bind variables in ATS to sections of the syntax tree instead of repeating them long form.

In theory, you could have a more complex type system that allowed for union types, etc. to solve this problem.

4 A-Normal Form

A-normal form is an intermediate representation for the compiler before it emits target environment code (for us, C).

In A-normal form, each function argument is simple and atomic– they must either be variables or values. That means that function arguments must be calculated and stored in a temporary variable, and then that variable can be passed into a function call.

[2]You can still create `timesFive` with these semantics, of course, but it's not as elegant. You'd have to do something like $\lambda x.multiply(tuple(5, x))$.

[3]During type checking; it does support them at runtime. There are some valid programs that don't type check.

For example, `foo(bar(1), bazz(2))` would become

```
    temp0 = bar(1)
    temp1 = bazz(2)
    temp2 = foo(temp0, temp1)
    return temp2
```

This intermediate representation makes it easy to emit code for destination languages with inconvenient or limited call semantics, like assembly.

Because in assembly we can only access immediate values, we can't call `foo` with complex arguments, only immediate values (eg stored in registers). In A-normal form, we can iterate over our variable bindings and emit each one as a set of assembly instructions without additional processing.

Fortunately for us, our destination language C supports calls like `foo(bar(1), bazz(2))`, so we can avoid this process. C will do it for us when we compile our generated C code.

4.1 A-Normal Form Datatype

You can represent a term in A-Normal form with these types:

```
Term: FunctionCall | Variable | Value
FunctionCall: (FunctionPointer, Arguments)
Arguments: (Variable | Value), Arguments | None
```

You'd need to convert each term in your AST into a `Term`, which (if it's a `FunctionCall` requires converting all of it's arguments into `Variable`s or `Value`s. You'd end up traversing the tree recursively , we did when type checking. At each step, if you had a function call, you'd recursively A-normalize its arguments. Then, you'd assign its return value to a `Variable`, so you'd be able to use it. This is another interpreter!

Here's some pseudocode:

```
a_normalize(term) =
case of term
| Variable(a) => let tmp_var = a
| Value(a) => a
| FunctionCall(function_pointer, arguments) =>
let another_tmp_var = a_normalize(arguments[0])
if len(arguments) > 1:
    a_normalize(arguments[1::]
let tmp_return = function_pointer(tmp_vars_we_have_from_recursive_a_norm_calls)
```

Obviously, this isn't complete pseudocode. However, you can see the recursive structure. We recurse down to the lowest level of our function calls, and assign each intermediate value to a temporary variable. Then, we use those temporary variables and immediate values to generate a AST that's in A-normal form.

Finally, when printing out C or Assembly code, you have an easy-to-traverse tree. At each step, you're just calling functions with immediate values (easy) or saving the results of function calls (also easy).

5 Code Emission

Like we did while evaluating with the closure-based interpreter, we emit code by traversing the AST. Our traversal function is of type $term \rightarrow (str, str)$. We process each term into an output (`out`) and a hoisted code block (`hoist`). If the term we're processing needs to generate any top-level code (eg. to generate the C function for a lambda), it outputs that to `hoist` and it's included in as a top-level declaration in the final output code. `out` should be an expression (not a statement) that will evaluate to the value of the input term.

This is a very powerful api; it allows us to use any number of statements to calculate our return value by putting them in a hoisted function that takes the context `c` and returns the value of the term. For example, in C if statements are statements and not expressions, so we can't use it in a block of code that's expecting an expression. We're able to generate and hoist a C function that uses an if statement and returns a value, and then `out` can be a call to this function, which is an expression. (We could also use a ternary operator.)

Essentially, `hoist` allows us to generate helper functions, which can be necessary for some terms, especially lambda expressions.

While processing a term, we will likely have to generate code for it's child terms (unless it's a primitive, like `TMint`). We can recursively call our same function to process those child terms, and substitute in `out` when we need the value they evaluate to. They might also generate hoisted code, so we always append that to whatever hoisted code we might have for this term and return that. This is valid because all hoisted code is top level, so we don't care about it's context– it can all sit at the root of the file.

6 Inner Functions and Closure Conversion

Much like in the closure-based interpreter, at run time lambdas take two arguments: the value of their argument, and the context in which to evaluate the body. Here's an example:

Input source code:

```
TMapp(
    TMlam("x", TMlam("y", TMadd(TMvar"x", TMvar"y"))),
    TMint(10)
)
```

Output generated C code:

```
#include "runtime_final.h"

lamval1
sym919(lamval1 y, Context c) {
  c = *extend_context(&c, "y", y);
  c = *extend_context(&c, "unused", close(sym919, c));
  lamval1 ret0;

  lamval1 tmp0 = LAMOPR_add(retrieve_variable(&c, "x"), retrieve_variable(&c, "y"));
  ret0 = tmp0;
  return ret0;
}


lamval1
sym225(lamval1 x, Context c) {
  c = *extend_context(&c, "x", x);
  c = *extend_context(&c, "unused", close(sym225, c));
  lamval1 ret0;

  lamval1 tmp0 = close(sym919, c);
  ret0 = tmp0;
  return ret0;
}


int main(void) {
```

```
21    Context c = *create_hash_map();
22    LAMVAL_print((((lamval1_clo) close(sym225, c))->fp)(LAMVAL_int(10), *(((lamval1_clo) close(sym225, c)
23  }
```

We begin by creating an empty context (line 21). Then, on line 22 we create a closure over `sym225` (our entry point) and our new empty context `c`.

In `sym225`, we extend the context to include our binding of `x`, and then we return a closure over our final inner lambda $\lambda y.x + y$

We ultimately print out a lambda closure for a function that adds 10. `x=10` is curried inside this function.

We can think about closures as containing instructions and a context. In C, we can't locally scope instructions this way, so we do the next best thing and hoist the C functions to the top level and keep track of a function pointer.

7 Conclusion

Lambda calculus is a powerful computational model. I've implemented a lambda to C compiler that is essentially made up of two interpreters: one that recursively traverses the AST and performs type-checking, and the other that recursively traverses the AST and emits C code.