Introduction to Lambda Calculus

Maciek Makowski (@mmakowski)

June 2, 2014

1 Motivation

Software is pervasive in the modern world and has influence over many aspects of our lives. In some cases, such as avionics or medical equipment control, human life depends on the correctness of software. Yet, high profile cases of bugs[1] do not inspire confidence in the state of software engineering. The "software crisis" is a phenomenon recognised by practitioners of the field. A number of ways to address the reliability issue has been proposed, from reliance on programmer's discipline[10][11], through tools that perform post-hoc validation of programs to ensure they do not contain suspicious coding patterns[12], to languages that restrict valid programs to ones whose properties can be formally proven. The latter approach relies on a body of theoretical knowledge that can appear intimidating. It turns out, however, that much of the required insight is built on systematic extensions of a very simple formal system – the lambda calculus. Familiarity with the fundamentals of lambda calculus is a prerequisite for proficiency with modern software engineering tools.

These notes aim to introduce lambda calculus in a very informal manner. The only background assumed is that of general programming experience.

2 Syntax

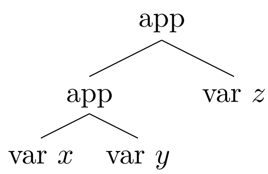
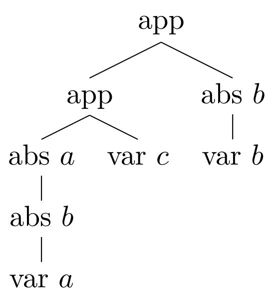
Given a set X of variables, terms of lambda calculus are generated by the following grammar:

$$
\begin{array}{llll}
\langle term \rangle & ::= & x & \text{(variable)} \\
 & | & (\lambda x.\langle term \rangle) & \text{(abstraction)} \\
 & | & (\langle term \rangle\ \langle term \rangle) & \text{(application)}
\end{array}
$$

[1] Infamous historical examples include Mars Climate Orbiter's inconsistent usage of units of measurement[6] and Therac-25 radiation therapy overdoses[7]. Recent faults such as security-related Apple goto fail[8] and OpenSSL Heartbleed bug[9], while not life-threatening, had wide-ranging implications for the security of e-commerce and privacy of internet users.

where $x \in X$. Notational convention is that application binds to the left, the full-stop can be treated as an opening parenthesis that extends until the end of the sub-term and redundant parentheses are omitted.

Example: shown below are sample lambda terms, on the left as they are typically written, on the right actual ASTs:

v_1	var v_1
$x\ y\ z$	
$(\lambda a.\lambda b.a)\ c\ (\lambda b.b)$	

Within a term, occurrences of variables that are not bound by enclosing abstraction – i.e. where the variable name does not appear in any abs node on the path to the root of the AST – are called *free*.

Example: free occurrences have been underlined in the sample terms below:

- $\lambda x.\underline{y}$
- $(\lambda a.\lambda \underline{b}.a)\ \underline{c}\ (\lambda b.b)$

Note that the same variable name might have both bound and free occurrences within a term, as in the second example. Terms with no free occurrences are known as *closed terms*, or *combinators*.

Summary: we have defined what lambda terms look like. The grammar has three productions – variable, abstraction and application – and generates abstract syntax trees.

3 Rewriting Rules

Lambda terms are not of much use without some operations that can be performed on them. Two[2] operations we will use going forward are presented below.

3.1 Renaming of bound variables

Intuitively, $\lambda x.x$ is similar to $\lambda y.y$ – the "shape" of these two terms is the same, they differ only in the choice of variable used. In practice we will often want to identify terms of the same structure. This intuitive similarity is captured by an operation called *α-conversion* that consistently renames variables. While the operation appears trivial, there are subtleties around bound vs. free variables – for instance, nested abstractions can use the same variable name – but these difficulties manifest themselves mostly in the implementation[3], so we will not worry about them in this presentation. The intuition about variable renaming is correct in most cases we are interested in. In particular, we can assume that all the variables in the terms we are working with have been chosen so that they are distinct. It is always possible to α-convert any term so that this is true.

Example: $(\lambda x.x\,y)\ (\lambda x.x) \longleftrightarrow_\alpha (\lambda a.a\,y)\ (\lambda b.b)$

3.2 Removal of abstraction under application

TODO: *β-reduction*

Example: $(\lambda x.x\,y)\ (\lambda z.z) \longrightarrow_\beta (\lambda z.z)\ y \longrightarrow_\beta y$

In the first reduction, $\lambda x.$ is eliminated by substituting $\lambda z.z$ for x, in the second, $\lambda z.$ is eliminated by substituting y for z.

This is the key operation of lambda calculus, as it represents a *computation.* It allows us to view a lambda term as a program that can be evaluated to a final value (i.e. a term that cannot be further reduced). That final value is called a *normal form* of a term.

An application that can be β-reduced is called a *redex.* There can be multiple redexes in a given term:

[2] The third frequently applied operation is introduction/removal of abstraction (η-conversion): $\lambda x.M\,x \longleftrightarrow_\eta M$ for any term M. We will not require it in this presentation of lambda calculus.

[3] For that reason a convenient way to represent lambda terms when implementing evaluation is positional (de Bruijn) encoding which replaces variable names with numerical index of the lambda that binds given variable occurrence. For example, de Bruijn representation of $\lambda a.\lambda b.a\ b\ c$ would be $\lambda.\lambda.1\ 0\ 2$ under naming context $\{c \mapsto 2\}$. The naming context is required to map free variables to indices.

$$(\lambda b.b)\,((\lambda c.\lambda d.d)\,\lambda f.f)$$

$$\overbrace{(\lambda a.\underbrace{(\lambda b.b)\,a})\,(\underbrace{(\lambda c.\lambda d.d)\,\lambda f.f})}$$

$$(\lambda a.a)\,((\lambda c.\lambda d.d)\,\lambda f.f) \qquad (\lambda a.(\lambda b.b)\,a)\,(\lambda d.d)$$

The choice of the redex to be reduced next is up to us. Prescribing which redex should be chosen gives rise to a reduction strategy. For example:

- TODO: call-by-name
- TODO: call-by-value

TODO: Church-Rosser

Summary: lambda-terms can be syntactically transformed according to two rules: α-conversion, which renames the variables, and β-reduction, which "applies" one term to another.

4 Mathematical Interpretation

So far we have discussed the syntax of lambda calculus without any mention of its semantics. The purpose of this restriction was to stress that all the constructs we will be building are defined as manipulation of term trees, and any meaning we might associate with them is secondary. That said, understanding the upcoming sections will be much easier with a mental model of what a lambda term represents. It perhaps comes as no surprise that a lambda abstraction can be interpreted as an anonymous function.

Example: in common mathematical notation we would write

$$f(x) = a * x + b$$

to describe a linear function of x. The same function can be represented by lambda term

$$\lambda x. + \,(*\, a\, x)\, b$$

where $*$ and $+$ are predefined binary functions that respectively multiply and add numbers and are written in prefix notation due to lambda calculus syntactic rules.

Lambda application is then simply application of the function represented by the first subterm to the argument represented by the second subterm. In this model, β-reduction is a method of function evaluation.

Summary: lambda abstractions can be treated as representing mathematical functions.

5 Programming

We claimed that a lambda term can be treated as a computer program. To substantiate this, let us see how familiar elements of programming languages can be represented in lambda calculus.

5.1 Conditionals

Choice of execution path to follow is a fundamental building block of most algorithms. It is usually represented as

$$\texttt{if}\ C\ \texttt{then}\ T\ \texttt{else}\ F$$

where C evaluates to either `true` or `false`, where `true` and `false` are some specific values chosen by us beforehand. If C evaluates to `true` then the whole conditional expression evaluates to the result of T, otherwise it evaluates to the result of F. To start with, let us choose some lambda-terms that will represent `true` and `false`:

$$\begin{aligned} \texttt{true} &= \lambda t.\lambda f.t \\ \texttt{false} &= \lambda t.\lambda f.f \end{aligned}$$

With these in place, the conditional expression can be written as

$$\begin{aligned} \texttt{test} &= \lambda c.\lambda t.\lambda f.c\,t\,f \\ \texttt{if}\,C\,\texttt{then}\,T\,\texttt{else}\,F &\equiv \texttt{test}\,C\,T\,F \end{aligned}$$

It is easy to see that indeed, if c is `true` then β-reduction of $c\,t\,f$ will yield t and if it is `false` then it will reduce to f.

5.2 Numbers

Another primitive essential in programming as we know it are numbers. After Peano, we can specify the set of natural numbers by means of a chosen value `0` and a function `succ` that maps every natural number to its successor. In lambda calculus these can be encoded as follows:

$$\begin{aligned} \mathtt{0} &= \lambda s.\lambda z.z \\ \mathtt{succ} &= \lambda n.\lambda s.\lambda z.s\,(n\,s\,z) \end{aligned}$$

This is what terms representing numbers look like with these definitions[4]:

$$\begin{aligned} &\mathtt{0} = & &\lambda s.\lambda z.z \\ &\mathtt{1} = \mathtt{succ}\ \mathtt{0} = & &\lambda s.\lambda z.s\,z \\ &\mathtt{2} = \mathtt{succ}\ \mathtt{1} = & &\lambda s.\lambda z.s\,(s\,z) \\ &\mathtt{3} = \mathtt{succ}\ \mathtt{2} = & &\lambda s.\lambda z.s\,(s\,(s\,z)) \\ &\vdots \\ &\mathtt{n} = & &\lambda s.\lambda z\,\underbrace{s\,(\ldots s\,(s}_{n}\ z)\ldots) \end{aligned}$$

Definition of arithmetics in this representation is reasonably straightforward in case of addition and multiplication:

$$\begin{aligned} \mathtt{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\ \mathtt{times} &= \lambda m.\lambda n.m\ (\mathtt{plus}\ n)\ \mathtt{0} \end{aligned}$$

Subtraction, however, turns out to be much more tricky to define[5].

5.3 Repeated Calculation

With numbers and conditionals in place, our language is still severly restricted. For example, how would we write a factorial function? As a reminder, the mathematical definition of factorial is

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n * (n-1)! & \text{otherwise.} \end{cases}$$

[4] Seeing how Peano arithmetic can be defined in lambda calculus, and drawing parallels with Zermelo-Fraenkel set-theoretical model of natural numbers and the build-out of other mathematical constructs on this basis, one could ask: can untyped lambda calculus be treated as a foundational theory in which all known mathematical concepts can be stated? Alonso Church had that in mind when he conceived lambda calculus, but it was soon proven by his studens, Kleene and Rosser[13], that in fact the calculus as we present it here is inconsistent, i.e. any proposition can be proven in it. TODO: why does it not concern us?

[5] If you attempt to do this as an exercise you might want to start with defining a function `pred` – the inverse of `succ`.

Implementation in a typical programming language involves either a recursion or a loop, neither of which is directly supported by lambda calculus. A way to deal with this is *fixed point operator*:

$$\mathtt{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

This combinator is easiest understood by watching it work in practice. Intuitively, we construct a function parameterised by a function, then use `Y` to feed the function into itself. For example factorial can be defined as follows[6]:

$$\begin{aligned} \mathtt{g} &= \lambda f.\lambda n.\mathtt{if\ eq}\ n\ \mathtt{0\ then\ 1\ else}\ (\mathtt{times\ n}\,(f\,(\mathtt{pred\ n}))) \\ \mathtt{factorial} &= \mathtt{Y\ g} \end{aligned}$$

To illustrate the working of `Y` let let us follow a single stage of evaluation of `factorial 3`:

$$\begin{array}{c} \mathtt{factorial\ 3} \\ \downarrow \\ \mathtt{Y\ g\ 3} \\ \downarrow \\ (\mathtt{h\ h})\ \mathtt{3} \\ \downarrow \\ \mathtt{g}\ (\mathtt{h\ h})\ \mathtt{3} \\ \downarrow \\ \lambda n.\mathtt{if\ eq}\ n\ \mathtt{0\ then\ 1\ else}\ (\mathtt{times\ n}\,(\mathtt{fct}\,(\mathtt{pred\ n})))\ \mathtt{3} \\ \downarrow \\ \mathtt{times\ 3}\ (\mathtt{fct\ 2}) \end{array}$$

$$(\mathtt{h\ h})\ \mathtt{2}$$

In general, a recursive function `f`, that in a language that supports direct recursion would be defined as

$$\mathtt{fun\ f} = M(\mathtt{f})$$

where $M(\mathtt{f})$ is the body of the function that contains reference to `f`, in lambda calculus can be defined as

$$\mathtt{f} = \mathtt{Y}\ (\lambda f.M(f))$$

6 Types

TODO

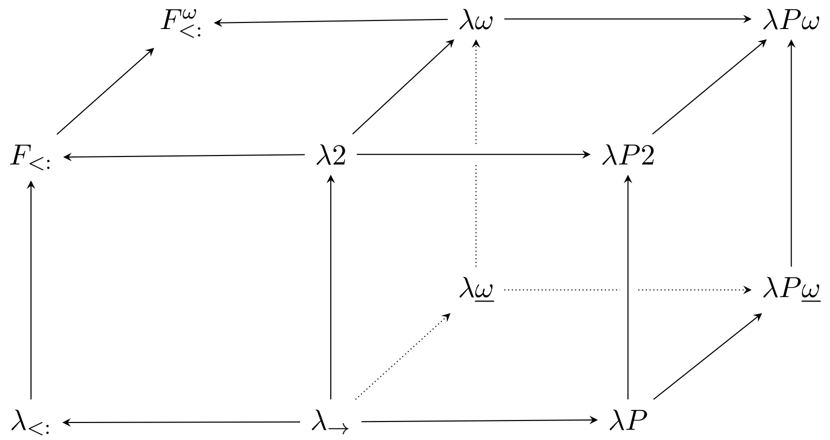
[6] Note that this definition is expressed entirely in lambda calculus. We have previously shown how to encode `if-then-else`, `times` and numbers, `pred` and `eq` can also be encoded with a little bit more machinery than what we managed to show in this short introduction – see e.g. [1] for full details.

7 Curry-Howard Correspondence

TODO

8 More Types

More elaborate calculi build on top of simply-typed lambda calculus. The diagram below shows some of the extensions.



Three of the direct extensions of simply-typed lambda calculus – $\lambda_{\to}$ in the diagram – are obtained by adding various forms of dependencies between types and terms:

- $\lambda 2$, also known as *System F*; adds terms depending on types, i.e. *polymorphism*
- $\lambda\underline{\omega}$; adds types depending on types, i.e. *type operators*
- λP; adds types depending on terms, i.e. *dependent types*

These extensions can be combined further into more powerful calculi. The eight calculi based on simple types and the three extensions mentioned form the *lambda cube*[4].

In addition, subtyping ($<:$) can be added to $\lambda_{\to}$, and a combination of this calculus with extensions of System F provides calculi in which object orientation can be modelled.

9 Further Reading

A direct inspiration for this talk was the presentation of lambda calculus in [1]. The book is very well written and builds a sophisticated type system in easy to follow steps, starting from untyped lambda calculus.

For a succinct but rigorous introduction to lambda calculus see [3].

TODO

References

[1] Benjamin C. Pierce, *Types and Programming Languages*, `http://www.cis.upenn.edu/~bcpierce/tapl/`

[2] Morten Heine B. Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7385`

[3] Henk Berendregt, Erik Barendsen, *Introduction to Lambda Calculus*, `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`

[4] Henk Berendregt *An Introduction to Generalized Type Systems*, `http://www.diku.dk/hjemmesider/ansatte/henglein/papers/barendregt1991.pdf`

[5] Klaus Grue, *Lambda calculus as a foundation of mathematics*, `http://www.diku.dk/~grue/papers/church/church.html`

[6] NASA, *Mars Climate Orbiter Team Finds Likely Cause of Loss*, `http://www.jpl.nasa.gov/news/releases/99/mcoloss1.html`

[7] Nancy Leveson, Clark S. Turner, *An Investigation of the Therac-25 Accidents* `http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html`

[8] *CVE-2014-1266*, `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266`

[9] *CVE-2014-0160*, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`

[10] Robert C. Martin, *Clean Code*

[11] various authors, *CERT C Coding Standard*, `https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=524435`

[12] Nick Rutar, Christian B. Almazan, Jeffrey S. Foster, *A Comparison of Bug Finding Tools for Java*, `http://www.cs.umd.edu/~jfoster/papers/issre04.pdf`

[13] Wikipedia, *Kleene-Rosser paradox*, `https://en.wikipedia.org/wiki/Kleene-Rosser_paradox`