Introduction to Lambda Calculus

Maciek Makowski (@mmakowski)

September 14, 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. Several ways of addressing the reliability issue have been proposed: from reliance on programmer's discipline[Clean Code][CERT], through tools that perform post-hoc validation of programs to ensure they do not contain suspicious coding patterns[CBFTJ], 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.

Arguably, familiarity with state-of-the-art tools is a matter of professionalism. Understanding of the basics of lambda calculus will help us better appreciate what the tools from formal methods end of the spectrum have to offer.

2 Syntax

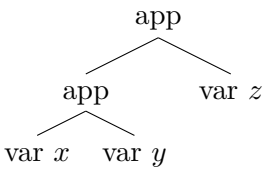
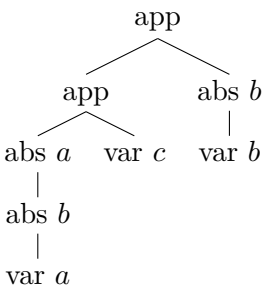
Given a set $\mathbb{X}$ of variables, terms of lambda calculus are generated by the following grammar:

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

[1] Infamous historical examples include Mars Climate Orbiter's inconsistent usage of units of measurement[MCO] and Therac-25 radiation therapy overdoses[Therac25]. Recent faults such as security-related Apple goto fail[CVE-2014-1266] and OpenSSL Heartbleed bug[CVE-2014-0160], while not life-threatening, had wide-ranging implications for the security of e-commerce and privacy of internet users.

where $x \in \mathbb{X}$. 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

The basic operation that modifies the structure of the term is β-*reduction*. It can be applied to every term or sub-term that is an application whose left-hand side is an abstraction:

$$(\lambda x.M)\ N \longrightarrow_\beta M[x/N]$$

where $M[x/N]$ is term M in which all free occurrences of x have been replaced by term N.

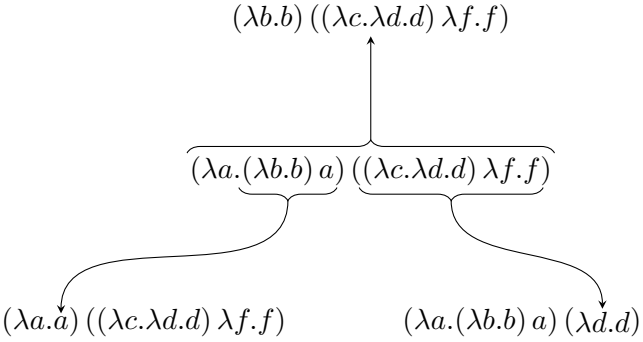
Example: $(\lambda x.x\, y)\ (\lambda z.z) \longrightarrow_\beta (\lambda z.z)\ y \longrightarrow_\beta y$
In the first reduction, $\lambda x.$ is eliminated by substituting $\lambda z.z$ for x, in the second, $\lambda z.$ is eliminated by substituting y for z.

[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.

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.

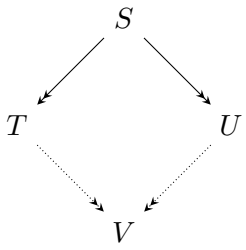
An application that can be β-reduced is called a *redex*. There can be multiple redexes in a given term:



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:

- restricting reduction to only the redexes that are not under abstraction and always starting with the innermost redex is *call-by-value* strategy;
- similarly restricting reduction to redexes that are not under abstraction, but starting with outermost redex is *call-by-name*[4].

Different evaluation strategies can stop at different terms; for example, if we use call-by-value then the chain of reductions will stop once there are no more redexes outside of abstractions. A more permissive strategy would allow the reductions to continue with those redexes under abstractions. Perhaps more interestingly, there are terms which loop forever under some evaluation strategies but reduce to a normal form under others. That said, unrestricted β-reduction is *confluent*: if a term S can be reduced in two different ways, producing terms T and U, then T and U can both be reduced to some common term V:



[4] Scala uses call-by-value evaluation strategy everywhere except for by-name function arguments (`arg: => T`) where call-by-name is used

This is formalised as *Church-Rosser theorem* and is an important result: it allows us to disregard the order of reductions when considering the semantics of a lambda term. Specifically, it justifies the interchangeability of lazy and eager evaluation.

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, the upcoming sections will be much easier to understand 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 interpreted as 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].

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`[6]:

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

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

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

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:

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

This is what terms representing numbers look like with these definitions[7]:

[5] One definition of functional programming is "programming with mathematical functions". By presenting how to program with lambda terms, which, we argued, represent such functions, we will demonstrate the basics of functional programming.

[6] Note that the syntax `name` $= T$ is not part of lambda calculus, it is just our way of giving meaningful names to terms. We can think of it as a preprocessor macro.

[7] 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[KRP], that in fact the calculus as we present it here is inconsistent, i.e. any proposition can be proven in it.

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

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

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

Subtraction, however, turns out to be much more tricky to define[8].

5.3 Repeated Calculation

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

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

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*:

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

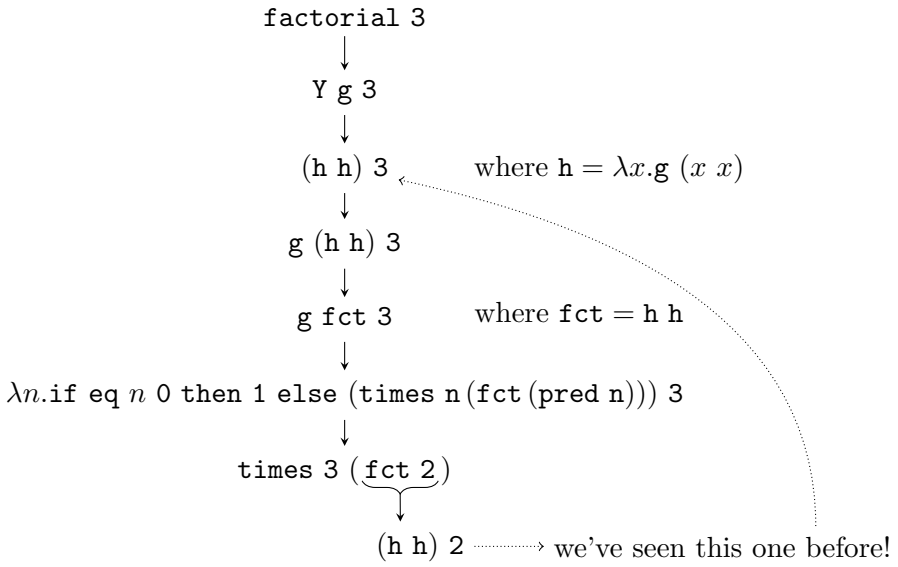
How does that help? Intuitively, we can construct a function parameterised by a function, then use `Y` to feed the function into itself. For example factorial can be defined as follows[9]:

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

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

[9] 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. [TAPL] for full details.

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



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

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

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

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

Summary: we can encode common programming constructs such as conditionals, numbers and recursion[10] as lambda terms.

6 Types

If $\mathbb{T}$ is the set of type variables then the grammar of types is:

$$\begin{aligned}\langle type\rangle &::= \sigma \\ &\mid \ (\langle type\rangle \rightarrow \langle type\rangle)\end{aligned}$$

[10]We have only shown how to encode a function that invokes itself recursively. Mutual recursion, i.e. a set of functions where each function can invoke each other in cyclic fashion can also be represented. We can construct a record of functions – at this point it might come as no surprise that a record can be encoded as a lambda abstraction – and apply `Y` to that record. See [TAPL] for details.

where $\sigma \in \mathbb{T}$. Notational convention is that arrow binds to the right, so we can write $\sigma \to \tau \to \rho$ instead of $(\sigma \to (\tau \to \rho))$.

Lambda terms can be assigned types by following the structure of the AST. We start with the leaves of the tree, i.e. variables, and make note of assumptions about their types; for example, $x : \sigma$ means that we assume variable x to have type σ. We then proceed up the tree, applying the following two rules to inner nodes:

$$\frac{M : \sigma \to \tau \qquad N : \sigma}{M\,N : \tau} \qquad \text{(application)}$$

$$\frac{\begin{array}{c}\cancel{x : \sigma} \\ \vdots \\ M : \tau\end{array}}{\lambda x.M : \sigma \to \tau} \qquad \text{(abstraction)}$$

In abstraction rule, we need to first make an assumption about the type of variable x that might occur free in term M. Once we establish the type of M to be τ based on this assumption, we use the assumed type as the type of argument of the abstraction, at which point the assumption is no longer needed – hence the crossing out.

As we proceed applying the rules, we might need to refine the assumptions. If we initially assumed $x : \sigma$ and $f : \tau$ but then come across application $f\,x$ then we know from the rule for application that the type of f must be an arrow type, so we have to refine the assumption to $f : \sigma \to \rho$.

Example: what is the type of $\lambda f.\lambda x.f(f\,x)$?

$$\dfrac{\dfrac{\dfrac{\cancel{f : \sigma \to \sigma} \qquad \dfrac{\cancel{f : \sigma \to \sigma} \qquad \cancel{x : \sigma}}{f\,x : \sigma}\,(app)}{f(f\,x) : \sigma}\,(app)}{\lambda x.f(f\,x) : \sigma \to \sigma}\,(abs)}{\lambda f.\lambda x.f(f\,x) : (\sigma \to \sigma) \to \sigma \to \sigma}\,(abs)$$

The first use of application rule eliminated the need for $x : \sigma$ assumption, the second eliminated $f : \sigma \to \sigma$.

We write $\Gamma \vdash M : \sigma$ to state that there exists a derivation with the set of assumptions (context) Γ that assigns type σ to term M. Notational convetion is that when the Γ is empty it can be omitted altogether, so for the example above we can write $\vdash \lambda f.\lambda x.f(f\,x) : (\sigma \to \sigma) \to \sigma \to \sigma$[11].

[11]This way of inferring types of terms has been introduced by Curry. An alternative system, by Church, requires explicit type declarations. These two calculi differ in some of their properties, but the type derivations in both are essentially equivalent[SU99].

Lambda calculus whose set of terms is restricted to terms that can be assigned types according to the rules above is called *simply-typed lambda calculus* and denoted by $\lambda_{\rightarrow}$. The typing restriction is severe: it only admits terms whose β-reduction terminates with a term that cannot be reduced further[12]. For example, recursion cannot be implemented in this calculus.[13]

Summary: lambda terms can be assigned types, which are built from type variables and arrows. If we insist that only terms that have a type are valid, we place limits on how the reduction of such terms can behave.

7 Curry-Howard Correspondence

In classical propositional logic every formula is either true or false. *Intuitionistic logic* is an alternative system, where a statement is true if it can be proven constructively[14] and false if a contradiction can be inferred from it. If neither of these is the case the truth of the statement remains unknown. A consequence of intuitionistic approach is that some of classical tautologies, such as the law of excluded middle, $p \vee \neg p$, cannot be proven.

Proofs in intuitionistic logic are trees with the formula to be proven in the root, axioms in the leaves and inner nodes containig formulae that can be inferred from children using a set of prescribed rules. As such, these proofs directly correspond to computations. As we argued in 3.2, so do lambda terms. Furthermore, types of simply-typed lambda calculus can be read as statements in the fragment of intuitionsitic logic that uses implication ($\rightarrow$) as the only logical connective. It turns out that the terms in this calculus can be interpreted as proofs of the propositions represented by their types. For a hint of how such proofs may look see the derivation tree for $\lambda f.\lambda x.f(f\,x) : (\sigma \rightarrow \sigma) \rightarrow \sigma \rightarrow \sigma$ in section 6.

The Curry-Howard correspondence does not end on $\lambda_{\rightarrow}$. It can be extended to a huge variety of type systems and logics and this fact has been exploited to transfer results between the fields of logic and computer science[15].

[12] This property is called *strong normalisation*.

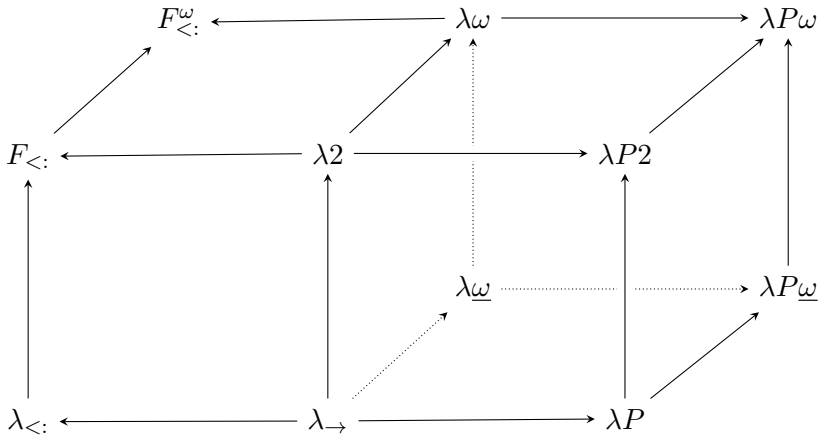
[13] This restriction can be seen as a good thing, since it eliminates a class of terms which we might not want to write, for example those that loop indefinitely, or as a bad thing, since it prevents us from writing useful functions such as the Fibonacci sequence. In practice this is the tension that programming language designers grapple with: provide a type system that eliminates as many undesirable programs as possible while eliminating as few desirable programs as possible.

[14] For this reason intuitionistic logic is also known as *constructive logic*.

[15] For an example of practical application of Curry-Howard correspondence see [Sabin11].

8 More Types

Simply-typed lambda calculus discussed in section 6 was very restrictive. More elaborate calculi build on top of simply-typed lambda calculus by providing extensions that allow typing of more terms. The diagram below shows some of the extensions.



Three of the direct extensions of simply-typed lambda calculus – $\lambda_{\rightarrow}$ 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*[Berendregt91].

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

9 Combinatory Logic

Let us define the following two combinators:

$$\mathtt{K} = \lambda x.\lambda y.x$$
$$\mathtt{S} = \lambda x.\lambda y.\lambda z.xz(yz)$$

First, notice that the identity function, $\lambda x.x$, can be expressed as $\mathtt{S\,K\,K}$. We will denote this as I. It turns out that any lambda term can be expressed as a sequence of applications of $\mathtt{S}$ and $\mathtt{K}$[16]. We can define a translation function T as follows:

$$
\begin{aligned}
T[\mathtt{K}] &= \mathtt{K} \\
T[\mathtt{S}] &= \mathtt{S} \\
T[x] &= x \\
T[(E_1\,E_2)] &= (T[E1]\,T[E2]) \\
T[\lambda x.E] &= \begin{cases} \text{if } x \text{ occurs free in } E: & \begin{cases} \text{if } E = x: & \mathtt{I}, \\ \text{if } E = \lambda y.E_1: & T[\lambda x.T[\lambda y.E_1]], \\ \text{if } E = (E_1\,E_2): & (\mathtt{S}\,T[\lambda x.E_1]\,T[\lambda x.E_2], \end{cases} \\ \text{otherwise:} & (\mathtt{K}\,T[E]). \end{cases}
\end{aligned}
$$

Example:

$$
\begin{aligned}
T[\lambda x.\lambda y.yx] &= T[\lambda x.T[\lambda y.yx]] \\
&= T[\lambda x.\mathtt{s}\,T[\lambda y.y]\,T[\lambda y.x]] \\
&= T[\lambda x.\mathtt{S\,I}\,(\mathtt{K}\,x)] \\
&= \mathtt{S}\,T[\lambda x.\mathtt{S\,I}]\,T[\lambda x.\mathtt{K}\,x] \\
&= \mathtt{S}\,(\mathtt{K}\,T[\mathtt{S\,I}])\,(\mathtt{S}\,T[\lambda x.\mathtt{K}]\,T[\lambda x.x]) \\
&= \mathtt{S}\,(\mathtt{K}\,(\mathtt{S\,I}))\,(\mathtt{S}\,(\mathtt{K\,K})\,\mathtt{I})
\end{aligned}
$$

The fact that any computable function can be expressed using only these two combinators is quite interesting in its own right, but we can do even better. Consider:

$$\mathtt{X} = \lambda x.x\,\mathtt{S\,K}$$

Now, we can express $\mathtt{S}$ and $\mathtt{K}$ using just $\mathtt{X}$:

$$
\begin{aligned}
\mathtt{K} &= \mathtt{X}\,(\mathtt{X}\,(\mathtt{X\,X})) \\
\mathtt{S} &= \mathtt{X}\,(\mathtt{X}\,(\mathtt{X}\,(\mathtt{X\,X})))
\end{aligned}
$$

That is, all computable functions can be expressed just as some sequence of applications of a single combinator.

[16] More precisely, for any lambda term T there exists an expression U that consist only of application of $\mathtt{S}$ and $\mathtt{K}$ that is *extensionally equivalent* to T, i.e. for any term V, application TV can be β-reduced to the same term as UV.

10 Further Reading

A direct inspiration for this talk was the presentation of lambda calculus in [TAPL]. 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 [BB00]. [SU99] contains a complete formal introduction to lambda calculus and an extensive material on Curry-Howard correspondence.

References

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

[SU99] 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`

[BB00] Henk Berendregt, Erik Barendsen, *Introduction to Lambda Calculus*, `http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf`

[Berendregt91] Henk Berendregt *An Introduction to Generalized Type Systems*, `http://www.diku.dk/hjemmesider/ansatte/henglein/papers/barendregt1991.pdf`

[LCSS] Henk Berendregt *The Lambda Calculus, its Syntax and Semantics*

[Sabin11] Miles Sabin, *Unboxed union types in Scala via the Curry-Howard isomorphism*, `http://www.chuusai.com/2011/06/09/scala-union-types-curry-howard/`

[Grue97] Klaus Grue, *Lambda calculus as a foundation of mathematics*, `http://www.diku.dk/~grue/papers/church/church.html`

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

[Therac25] Nancy Leveson, Clark S. Turner, *An Investigation of the Therac-25 Accidents* `http://courses.cs.vt.edu/cs3604/lib/Therac_25/Therac_1.html`

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

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

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

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

[CBFTJ] 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`

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