Propositions and Types, Proofs and Programs

Part III: Lambda Calculus

Ranald Clouston

School of Computing

Australian National University

ANU Logic Summer School 2023

Universal Models of Computation

Michael Norrish's lectures showed how the limits of mathematics and computation were startlingly revealed in the 1920s and 30s.

As part of this effort, a variety of universal models of computation were developed claiming to be able to express all algorithms.

- All serious efforts proved to be equally powerful
- No algorithm yet found not expressible by such models
- Turing machines, Gödel's recursive functions…
- The oldest proposed model: the lambda calculus of **Alonzo Church** (1903-1995)

Photo c/o MacTutor History of Mathematics Archive

The Three Ingredients of the Lambda Calculus

Variables: Usually written with letters like `x,y,z`

- Like the unknowns of mathematics or functional programming, not the memory addresses of imperative programming.

Lambda Abstractions: given a term (program) `t`, we have `λx.t`

- Intuition: wait for an input, replace all occurrences of `x` in `t` with it
- (Informal) example: `λx.x+3` takes one input, and adds 3 to it

Application: given terms `t` and `u`, we have `t u`

- Intuition: give `u` as input to `t`

Variable Binding

One can spill a lot of ink getting this formally correct, but it is important to understand the distinction between **free** and **bound** variables.

In ordinary mathematics, you would understand ‘x=3’ to be a statement about a variable x, presumably introduced earlier.

- Such an x is called *free* in the mathematical fragment x=3

On the other hand, if you read:

> Consider an integer x. It is bigger than 2 but smaller than 4. Hence x=3.

You would *not* consider this to be a statement about a variable called x that does not apply to some other variable called y. The name chosen is irrelevant to the truth of the statement.

- Such an x is *bound*

Variable Binding in the Lambda Calculus

The lambda calculus also has a notion of free and bound names

- Indeed, so do most programming languages

The variable `x` (and no others) is free in the variable `x`.

If the variable `x` is free in `t` or `u`, then it is free in the application `t u`.

But λ is a **binder**: `x` is *not* free in `λx.t`

- Any other free variable in t remains free in `λx.t`

A **closed** (or ‘complete’) lambda-calculus program will have no free terms, but is built out of subterms that do have free variables.

The Power of the Lambda Calculus

By clever **encodings** one can capture all mathematics with lambda calculus terms.

- e.g. `λf.λx.f(fx)` can encode the number 2
- Why? Because it works! Different encodings also work
- Analogy: encoding data and programs as binary on a digital computer

Lambda calculus terms can then run via one rule, called **beta-reduction**

- `(λx.t)u ↦ t[u/x]` (`t` with all free occurrences of x replaced by u)
- e.g. `(λx.x+3)2` first substitutes 2 for x to yield 2+3
- All of computation becomes available! But not so pleasant to use…

The Lambda Calculus is Untyped

You can do some not-very-sensible things in the lambda calculus

- Giving functions the wrong number of inputs
- Breaking your encodings, e.g. giving 3 as an input to 2
- Giving functions as inputs to themselves

Some programming languages are like this!

- Recommended short video in the link

But many languages prevent you from writing various sorts of garbage, by implementing **types**.

The History of Types

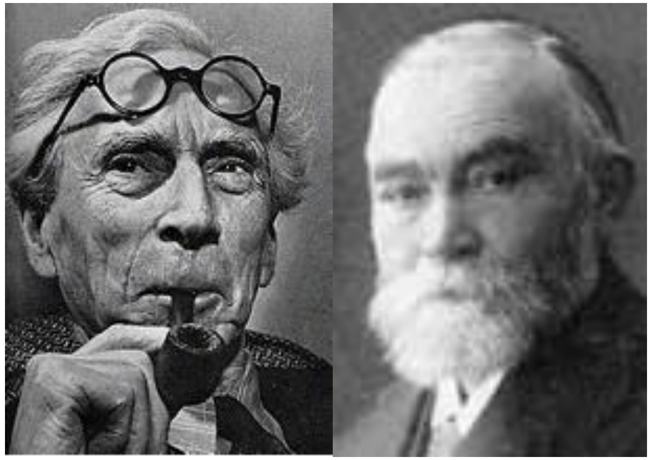
Types arose not in computing, but in the foundations of mathematics.

In 1902 **Bertrand Russell** (1872-1970) discovered an inconsistency in the work of **Gottlob Frege** (1848-1925), now called **Russell's paradox**:

- Some sets are members of themselves (e.g. the set of all sets)
- But Frege's system allowed the definition of 'the set of all sets that are not members of themselves'. Is this set inside itself?

Russell and others developed types to make certain definitions 'illegal'

- Like implementing a type system for the lambda-calculus, or a programming language – previously acceptable definitions become unacceptable.



Photos c/o Stanford Encyclopedia of Philosophy (1, 2)

Simply Typed Lambda Calculus

Church introduced types for the lambda calculus to create the simply typed lambda calculus (STLC) in 1940

- ‘Simply’ here implies that more complicated notions of type exist
- Original motivation was encoding logical quantifiers that bind names

This rules out certain untyped lambda calculus programs

- Wrecks all the careful encodings of mathematics in the untyped system!
- So if we want to regain some mathematics, will have to build back up to it with specific types for e.g. natural numbers…
- No longer a universal model of computation
- (but according to Barendregt and Barendsen: “in order to find … computable functions that cannot be represented, one has to stand on one’s head”)

The Types of STLC

The fundamental notion of the lambda calculus is that of *function*, so we lead with **function types**

- If A and B are types, then so is A→B
- (haven't we used that symbol before?)

To make this work we need a set of base types `b`,`c`…

- You can imagine these are useful types like `Nat`, `Bool`… but anything will do
- We hence have types like `b→b`, `b→c`, `b→(c→b)`, `(b→b)→b` …

In fact these are the *only* types of the minimal STLC, but we will find it convenient to add more soon.

The Type of Variables

All STLC terms (including ‘incomplete programs’ with free names) should have a type.

But what is the type of a variable x?

Could be anything, so we need to explicitly record it

- Could record it with the variable, e.g. x^A to mean ‘x has type A’
- More usual to type free names via a **typing context**, a set x:A,y:B,… of variable-type pairs where no variable appears twice.

Hence the **typing rule**:

$$\frac{}{\Gamma,\ x{:}A \vdash x{:}A}$$

Haven’t we seen this before?

The Type of Variables

All STLC terms (including ‘incomplete programs’ with free names) should have a type.

But what is the type of a variable x?

Could be anything, so we need to explicitly record it

- Could record it with the variable, e.g. x^A to mean ‘x has type A’
- More usual to type free names via a **typing context**, a set x:A,y:B,… of variable-type pairs where no variable appears twice.

Hence the **typing rule**:

$$\frac{}{\Gamma,\ A \vdash A}\ \text{AX}$$

The Type of Applications

Application is supposed to mean an input being given to a function.

A function should have function type, $A \to B$.

A legal input should have exactly the type A.

And its output should have type B.

Hence the typing rule:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash ft : B}$$

You definitely know this rule!

The Type of Lambda Abstractions

Say we have a term t of type B, possibly including a free variable x of type A.

We can turn this into a function that accepts inputs of type A, and gives outputs of type B.

Rule:

$$\frac{\Gamma, x{:}A \vdash t{:}B}{\Gamma \vdash \lambda x^{A}.t{:}A \to B}$$

Isn’t this staggering?

- But first, a fussy note about notation…

A Note on Lambda Abstraction Notation

We wrote $\lambda x^{A}.t$, explicitly recording the type of the bound variable.

- x is 'killed' from the context but we might need to remember its type
- This obviously was not done for the untyped lambda calculus
- Motivation: to distinguish e.g. $\lambda x^{Bool}.x$ from $\lambda x^{Nat}.x$
- These are different programs, because no polymorphism (yet!)

But we will usually omit the type from lambdas where there is no potential for confusion.:

$$\frac{\Gamma, x{:}A \vdash t{:}B}{\Gamma \vdash \lambda x.t{:}A \to B}$$

Meet Curry and Howard (At Last!)

This rhyme / pun / coincidence between implication and functions was noted by **Haskell Curry** (1900-82) in 1934

- Using the *SKI combinator calculus* rather than lambda calculus
- Coincidence considered amusing rather than important

Things took off in 1969 when **William Howard** (1926-) developed the deep connection between computation and proof normalisation

- So Prawitz's 1965 work key here

Photos c/o Haskell Wiki and Wadler's blog