

## Comp 411 Notes - 26 Jan 2005

The Untyped Lambda Calculus

Reproduced by Eric Cheng  
(ericc@rice.edu)

### 1 Introduction

In this class, we discussed some language features that exist in many languages and the practicalities of carrying out type proofs for them. We also discussed briefly what language features the pure untyped lambda calculus could emulate. Finally, we introduced the standard notion of reduction for the lambda calculus.

### 2 Introducing the Untyped Lambda Calculus

Type systems help us perform formal proofs for safety and correctness of a language. The language we have been discussing was simple but very limited in its usefulness (Is the language Turing complete?). It would be inappropriate to focus the rest of our development on small and arbitrary language like the one that we've looked at so far. This leads to the systematic critique of the language that we looked at before (in that it is limited), and that is what leads to the list of features that we collect.

**Exercise:** Is the language `arith` Turing complete?

What are some of the fancy features for untyped languages that people would like to see?

- Loops
- Recursion
- Side effects / state change
- Variables
- Records, data structures (+fixins)
- Functions: first-class functions (economic-class/business-class functions)
- Goto / jumps (Exceptions, control constructs such as 'switch')
- Objects
- Classes
- Dynamically allocated memory (related to data structures)

- Pointers<sup>1</sup>

However, the feature set is too big for us to carry out safety proofs for it. We would like to define a language with a smaller set of features. The reasons are:

- It takes less time to prove, but we still want to capture interesting features to save time.
- For some of the above features, it is simply not yet known how to extend certain type systems to support them. For examples, extending dependent type systems to deal with effects, objects or exceptions, are problems being actively investigated.
- Nearly all features can be emulated by the smaller set of features we define.

**Exercise:** Look up on the web for Turing completeness of LC and share your findings on the mailing list.

The language we will now discuss was introduced by Alonzo Church and his co-workers in the 1920s and '30s (Church, 1941).

$$\begin{aligned} e & ::= x | \lambda x. e | e e \\ v & ::= \lambda x. e \end{aligned}$$

For example, we saw in COMP 311 how some of these features can be translated into (or emulated in) the LC:

- Recursion: Y combinators.
- Data structures: Church encodings.
- Exceptions, goto, and jumps: continuations.
- Objects/classes: ?? (not completely sure).

In the course, we will also occasionally extend the basic lambda calculus with some of these features to make sure that the proof techniques that we cover are in fact sufficient to treat these features.

The questions we want to ask now is “what is the semantics for this language?”

Let’s first look at the syntax of the language.

Lambda calculus term	Corresponding mathematical term
$id \equiv \lambda x. x$	$f(x) = x$
$s \equiv \lambda f. \lambda g. \lambda x. g(f(x))$	$g \circ f = x \mapsto g(f(x))$

Table 1: Lambda terms in terms of their mathematical notation.

---

<sup>1</sup>The translation of pointers into an LC concept is a fairly non-trivial issue, and active research on this issue can be found in the area of denotational semantics of languages with pointers.

Reduction semantics:

$$\begin{aligned} id &\not\rightarrow_{\beta} \\ id\ s \equiv (\lambda x.x)s &\rightarrow_{\beta} s \end{aligned}$$

Table 2: Examples of reductions for the lambda calculus.

The first example illustrates that lambda terms reduces to nothing.  $\beta$  here means “reduction semantics.” There are two types of reduction semantics, namely, Call By Name (CBN,  $\beta$ ) and Call By Value (CBV,  $\beta_v$ ) (Plotkin, 1975).

The second examples shows how the identity function applies to terms. How do we generalize this schema?

$$(\lambda x.\underbrace{x}_{e_1})\underbrace{s}_{e_2} \rightarrow_{\beta} \underbrace{s}_{e_3}$$

Here, since we cannot reduce  $e_2$ , it is applied to the identity function, which takes an argument and returns it as is. That is,  $e_1$  gets returned, which is actually  $s$ , and so  $e_3$  is still  $s$ .

But this is only true for the identity function. How do we reason the cases for all other functions? What we need here is a way to ‘substitute’ terms:

$$e_1[x := e_2] \text{ (read: } e_1 \text{ with } x \text{ substituted to } e_2\text{)}$$

In general, the **standard notion of reduction for the lambda calculus** is:

$$\begin{aligned} (\lambda x.e_1)e_2 &\rightarrow_{\beta} e_1[x := e_2] \text{ (sound in CBN, unsound in CBV)} \\ (\lambda x.e_1)v &\rightarrow_{\beta_v} e_1[x := v] \text{ (sound in CBV)} \end{aligned}$$

However, how to perform substitution is not defined yet. There are a lot of details that go into the substitution process. The most important issue is **variables**, which we will discuss in the next lecture.

### 3 Bibliography

- Church, Alonzo. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- G. Plotkin. *Call-by-name, call-by-value and the lambda calculus*. Theoretical Computer Science, 1:125-159, 1975.

## 4 Trivia

- “Scribber” is not a word. The p.p. form for ‘scribe’ is ‘scribed’.