Type theory

Simon Charlow (simon.charlow@rutgers.edu)

September 30, 2015

1 Lambda calculus, redux

Why we bother:

- λs let us build functions **anonymously**.
- Simplification of λ-terms is a straightforward way to compute values for complex expressions.

1.1 Syntax

Syntax: the set of λ-terms, Λ, is the smallest set meeting the following conditions:[1]

1. (**Variables**) If x is a variable, $x \in \Lambda$.
2. (**Abstraction**) If x is a variable and $M \in \Lambda$, then $(\lambda x.\, M) \in \Lambda$.
3. (**Application**) If $M, N \in \Lambda$, then $(M\, N) \in \Lambda$.

And that is literally it. But some abbreviatory conventions improve usability:

1. Application is left-associative. For example:
   - 'M N L' instead of '((M N) L)'
2. Parentheses can be omitted whenever doing so doesn't create ambiguity. For example:
   - 'λx. M N' instead of 'λx. (M N)'
   - 'λx. M' instead of '(λx. M)'
   - 'λx. λy. M' instead of '(λx. (λy. M))'

   NB: λx. M y is equivalent to λx. (M y) and (λx. (M y)), but *not* (λx. M) y. I.e., the scope of a λ always extends as far to the right as possible (i.e. while still yielding a well-formed term).
3. A sequence of abstractions can be abbreviated. For example:
   - 'λxyz. M' instead of 'λx. λy. λz. M'

I recommend mastering at least the first two conventions. Whether you use the third is up to you…

[1] As we saw in the last class, in BNF this amounts to $E ::= V \mid (\lambda V.\, E) \mid (E_1\, E_2)$.

1.2 "Semantics"

Core equivalences:

- (**Renaming bound variables**) $\lambda x.\, M \underset{\alpha}{\longrightarrow} \lambda y.\, M[x \to y]$
- (**Applying functions to arguments**) $(\lambda x.\, M)\, y \underset{\beta}{\longrightarrow} M[x \to y]$
- (**Extensionality**) $f \underset{\eta}{\longrightarrow} \lambda x.\, f\, x$

Two of these equivalences rely on the notion of **substitution**, defined as follows:

> $M[x \to y]$ is the formula you get by replacing all M's **free** occurrences of x with y.

The **free** variables in an expression E ('$\mathsf{FV}\, E$') are the variables in E not **bound** by a λ:

$$\begin{aligned} \mathsf{FV}\, x &= \{x\} \\ \mathsf{FV}\, (\lambda x.\, M) &= \mathsf{FV}\, M - \{x\} \\ \mathsf{FV}\, (M\, N) &= \mathsf{FV}\, M \cup \mathsf{FV}\, N \end{aligned}$$

In other words, you can read $M[x \to y]$ as "M, but where the (free) x's becomes y's".

Some examples of substitution in action:

$$\begin{aligned} x[x \to y] &= y \\ (\lambda x.\, x)[x \to y] &= \lambda x.\, x \\ (\lambda f.\, f\, x)[x \to y] &= \lambda f.\, f\, y \end{aligned}$$

Substition has various notations. Sometimes you see '$M[x \coloneqq y]$'. Sometimes, as in our last class, you see '$M[y \,/\, x]$'. I'm hoping the present choice will be the most perspicuous. Let me know.

A couple examples of α, β, and η in action (notice that α-, β-, and η- equivalence can apply to proper *sub-formulas* of any λ-expression, as well as maximal λ-expressions):

$$\lambda f.\, \lambda x.\, f\, x \underset{\alpha}{\longrightarrow} \lambda f.\, \lambda z.\, f\, z \underset{\alpha}{\longrightarrow} \lambda g.\, \lambda z.\, g\, z$$

$$(\lambda f.\, \lambda x.\, f\, x)\, (\lambda y.\, y) \underset{\beta}{\longrightarrow} \lambda x.\, (\lambda y.\, y)\, x \underset{\beta}{\longrightarrow} \lambda x.\, x$$

$$\lambda f.\, f \underset{\eta}{\longrightarrow} \lambda f.\, \lambda x.\, f\, x \underset{\eta}{\longrightarrow} \lambda f.\, \lambda x.\, \lambda y.\, f\, x\, y$$

A β-reduction is **invalid** if it places a variable within the scope of a co-indexed λ:

$$(\lambda x.\, \lambda y.\, f\, x\, y)\, y \underset{\beta?\ \textbf{NO.}}{\longrightarrow} \lambda y.\, f\, y\, y$$

To avoid this unfortunate situation, exploit α-equivalence:

$$(\lambda x.\, \lambda y.\, f\, x\, y)\, y \underset{\alpha}{\longrightarrow} (\lambda x.\, \lambda z.\, f\, x\, z)\, y \underset{\alpha}{\longrightarrow} \lambda z.\, f\, y\, z$$

An α-equivalence is **invalid** for an expression E if it binds a free variable in E:

$$\lambda x.\, x\, y \underset{\alpha?\ \textbf{NO.}}{\longrightarrow} \lambda y.\, y\, y$$

To avoid this unfortunate situation… Don't get into it!

2 Types

The variety of the λ-calculus defined in the last section is **untyped**. Because of this, you can define functions like the following:

$$\omega \coloneqq \lambda x.\, x\, x$$

Which you can use to define the following:

$$\Omega \coloneqq \omega\, \omega$$

As we saw in the last class, Ω has an interesting property: β-"reducing" it gives you exactly what you already had!!

$$\Omega \underset{\beta}{\longrightarrow} \Omega$$

We'll be working with a different system in this class, called the **simply-typed** λ-calculus.

What are **types**? Pretty simple: types are nothing more than **sets of objects**! For example, the type $\mathbb{N}$ is the set of natural numbers.

In semantics, we work with two basic types: the type of individuals (roughly, things that can go in subject or object position[2]), and the type of truth values. By convention, the type of individuals is written 'e' (for 'entity', probably), and the type of truth values is written 't'.

$$\mathtt{e} ::= \{x : x \text{ is an individual}\}$$
$$\mathtt{t} ::= \{0, 1\}$$

Using our basic types, we can give an inductive definition for the set of **all types** τ, as follows:

1. $\mathtt{e}, \mathtt{t} \in \tau$.
2. If $\alpha, \beta \in \tau$, then $\langle \alpha, \beta \rangle \in \tau$.
3. Nothing else is in τ.

Some examples of types according to this "grammar":

- $\mathtt{e}$
- $\langle \mathtt{e}, \mathtt{t} \rangle$
- $\langle \langle \mathtt{e}, \mathtt{t} \rangle, \mathtt{t} \rangle$
- $\langle \langle \mathtt{e}, \mathtt{e} \rangle, \langle \langle \mathtt{e}, \mathtt{e} \rangle, \langle \mathtt{e}, \mathtt{e} \rangle \rangle \rangle$

The simple types are the individuals and the truth values. What are the complex types? The types of **functions**: $\langle \alpha, \beta \rangle$ is the type of functions from things of type α to things of type β.

An example. Let's begin with the set of linguists. This set has a characteristic function, which we might name LINGUIST. This function can be represented in λ-notation as follows (notice that this is just the η-expansion of LINGUIST!):

$$\lambda x.\, \textsc{linguist}\, x$$

[2] NB: semanticists think of lots of things as individuals, including numbers, tables and chairs, pets, bacteria, and so on.

This function takes individuals as arguments and returns truth values. Accordingly, its type is $\langle e, t \rangle$.

Likewise, an $\langle \langle e, t \rangle, t \rangle$ function takes functions from individuals to truth values as arguments, and returns truth values. A trivial example is the following:

$$\lambda P.\, 1$$

In the simply-typed λ-calculus, every function is **partial**. That is, every function expects its arguments to be of a certain type, and doesn't know what to do otherwise. For example, $\lambda x.\, \textsc{linguist}\, x$ wouldn't know what to do if you fed it $\lambda x.\, \textsc{philosopher}\, x$. The result would be **undefined**.

There are a number of ways to notate restrictions on types. Heim & Kratzer use the following convention (more generally, they use the ':' notation to indicate partiality):

$$\lambda x : x \in e.\, \textsc{linguist}\, x$$

Another way to go (the way I prefer, maybe because it saves symbols), is to use subscripts:

$$\lambda x_e.\, \textsc{linguist}\, x$$

In any case, type notations are often omitted when context makes clear what an expression's type is.

Some other examples:

- $[\![\text{blue}]\!] = \lambda x_e.\, \textsc{blue}\, x$
- $[\![\text{devoured}]\!] = \lambda x_e.\, \lambda y_e.\, \textsc{devoured}\, x\, y$
- $[\![\text{not}]\!] = \lambda p_t.\, 1 - p$
- $[\![\text{and}]\!] = \lambda p_t.\, \lambda q_t.\, p = q = 1$

In order:

- The type of a simple adjective is a function from individuals to truth values, type $\langle e, t \rangle$.
- The type of a transitive verb is a function from individuals to: a function from individuals to truth values, type $\langle e, \langle e, t \rangle \rangle$.
- The type of sentential negation is a function from truth values to truth values, type $\langle t, t \rangle$.
- The type of sentential conjunction is a function from truth values to: a function from truth values to truth values, type $\langle t, \langle t, t \rangle \rangle$.

In this manner of speaking, we can talk about denotations having types, as well as syntactic units (i.e., the type associated with a piece of syntax's denotation). We can use the following notation to indicate that an expression has a certain type:

$$\text{devoured} :: \langle e, \langle e, t \rangle \rangle$$

At the risk of burdening you with another notational convention, I'll mention that the way I prefer to write types in my own work (because it's way easier to parse than the brackets) is using arrows:[3]

$$\begin{aligned} \lfloor e \rfloor &= e \\ \lfloor e \to t \rfloor &= e \to t \\ \lfloor \langle e, \langle e, t \rangle \rangle \rfloor &= e \to e \to t \\ \lfloor \langle \langle e, t \rangle, t \rangle \rfloor &= (e \to t) \to t \end{aligned}$$

[3] In fact, you could *omit* the arrows entirely and still have a perfectly good notation scheme for types. But that's getting a bit telegraphic for most people's tastes.

Notice that in the simply-typed λ-calculus, it **makes no sense** for a function to apply to itself:

- Take an arbitrary f. Since f is a function, it will have type $\langle \alpha, \beta \rangle$, for some types α and β.
- But then, for $f\,f$ to be defined, f needs to be able to take itself as an argument.
- So then the functor f's type must be $\langle \langle \alpha, \beta \rangle, \beta \rangle$.
- But then the argument f's type must also be $\langle \langle \alpha, \beta \rangle, \beta \rangle$.
- In which case f's type must actually be $\langle \langle \langle \alpha, \beta \rangle, \beta \rangle, \beta \rangle$.
- And so on…

This means that ω, which crucially involves self-application, is impossible to define in a simply-typed system. Ergo, Ω is impossible to define.

In fact, every simply-typed λ-term is guaranteed to **terminate** into a unique **normal form**, in which no further β-reductions are possible. Because every expression has a *unique* normal form, the order of reductions never makes any difference. For example:

$$\begin{aligned} (\lambda x.\, f\,x)\,((\lambda y.\, g\,y)\,z) &\xrightarrow[\beta]{} (\lambda x.\, f\,x)\,(g\,z) \xrightarrow[\beta]{} f\,(g\,z) \\ (\lambda x.\, f\,x)\,((\lambda y.\, g\,y)\,z) &\xrightarrow[\beta]{} f\,((\lambda y.\, g\,y)\,z) \xrightarrow[\beta]{} f\,(g\,z) \end{aligned} \tag{1}$$

Unlike ω, it is possible to assign types to all the expressions in this term in a way that achieves a defined end result. Exercise: come up with one such typing scheme.