Ling 130 Notes: The λ-Calculus: Part 1

James Pustejovsky

February 4, 2010

1 Defining Functions

In this handout we will examine the ways in which functions are defined in a number of languages that you are no doubt familiar with. We will then give the basic syntax for the λ-calculus and move on to the rules of application and abstraction for the language.

The λ-calculus is a formal notation, theory, and a model of computation.[1] The λ-calculus allows you to define functions without giving them names, that is, they are anonymous. Expressions defined in this notation are called λ-expressions, and every λ-expression denotes a function under some interpretation.

A λ-expression consists of three kinds of terms:

(a.) **variables:** x**,**y**,** z**, etc.:**
Variables can be *bound* or *free*. Variables are bound to functions in an environment, and they become bound by usage in an abstraction.

(b.) **abstractions:** $\lambda V.E$:
V is a bound variable over the body E. The expression denotes the function that, when given an argument A, evaluates to the function E', with all occurrences of V in E replaced with a, written as $E[a/V]$.

(c.) **applications:**(E_1E_2):
If E_1 and E_2 are λ-expressions, so is (E_1E_2). Application is function evaluation, where we apply the function E_1 (the rator) to the argument E_2 (the rand).

Specified in BNF, the formal syntax of the λ-Calculus is quite simple:

$$E ::= V \mid \lambda V.E \mid (E_1E_2)$$

$$V ::= x \mid y \mid z \mid \ldots$$

1.1 Conversion and Reduction Rules

Expressions can be transformed into equivalent forms through the application of three rule schemas.

(a.) α**-conversion**:
Any abstraction $\lambda V.E$ can be converted to $\lambda V.E[V'/V]$ iff $[V'/V]$ in E is valid.

[1] Prof. Mairson's graduate class presents a lot of interesting properties of the λ-calculus, and I encourage you to attend his class when it is offered, if you are interested in functional programming or polymorphic type systems.

(b.) β**-conversion**:
Any application $(\lambda V.E_1)E_2$ can be converted to $E_1[E_2/V]$ iff $[E_2/V]$ in E_1 is valid.

(c.) η**-conversion**:
Any abstraction $(\lambda V.E)V$ where V has no free occurrences in E can be converted to E.

1.2 Syntactic Sugar

The λ-calculus binds variables in a way similar to the sugaring that the language Scheme establishes local variable binding. Scheme has `let` and `let*` operators for establishing local bindings of variables to values over a lexically scoped block. In a `let` expression, all values in the list of bindings are evaluated and then bound to the local variables. In a `let*` expression, the values are evaluated and bound to the variables sequentially.

```
(define x 2) => x
(let ((x 3) (y x)) (* x y)) => 6
(let* ((x 3) (y x)) (* x y)) => 9
```

This is a sugared version of:

```
((lambda (x y) (* x y)) 3 x) => 6
((lambda (x) ((lambda (y) (* x y)) x)) 3) => 9
```

2 The Simply Typed λ-Calculus

We will now present a language which uses specific types of entities, where the entities are combined with the λ-calculus. From a nonempty set $BasType$ of basic types, the set T or types is the smallest set such that:

(a) $BasTyp \subseteq T$

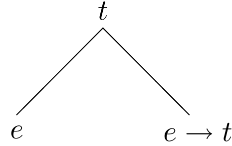
(b) If $\sigma, \tau \in T$, then $\sigma \to \tau \in T$.

We will take $BasTyp$, following Montague, to be

$$BasTyp = \{Ind, Bool\} \text{ or } \{e, t\}$$

where e is the type of individuals and t is the type of propositions.

In order to demonstrate the application and abstraction of typed expressions, let us consider some examples from linguistic expressions. Given the basic types of e and t, we can construct recursively other types, which in fact correspond quite nicely to notions that we are familiar with from first order logic or semantics. Consider the following. If an expression such as "John", denoting the individual John in the world, is an entity and hence of type e, and an expression such as "John fell" is a proposition that is either true or false (and hence of type t), then what is the predicate "fell" in this expression typed as? Well, if the mode of construction is functional, then, it will be a function, typed as $e \to t$.



The types here are acting functionally, according to the rule of application, given below:

(a) Function Application: If α is of type a, and β is of type $a \rightarrow b$, then $\beta(\alpha)$ is of type b.

Hence, the type for an intransitive verb is $e \rightarrow t$. In fact, this is the type for any predicate in general.

(i) walk: $e \rightarrow t$

(ii) walk slowly: $e \rightarrow t$

(iii) eat fish: $e \rightarrow t$

(iv) tell Mary where to buy her house: $e \rightarrow t$

The rule for abstraction allows us to substitute a predicate constant for a λ-expression of the appropriate type:

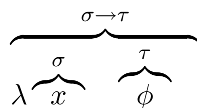
(a) $walk = \lambda x[walk(x)]$

(b) $love = \lambda x \lambda y[love(y,x)]$

Notice that a binary relation is yet another function, built out of the simpler function of an intransitive predicate.

2.1 Determining the Type of an Expression

For any expression ϕ, of type τ, the functional abstraction of ϕ with variable x of type σ has the type $\sigma rightarrow \tau$. This is visualized below:



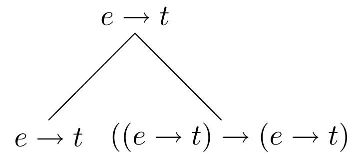
So, it might be clear how to make unary, binary, and ternary relations using functional abstraction and types, but what do we do for cases that appear to be higher order in nature? For example, consider the behavior of adverbs.

(a) John fell slowly.

(b) Mary talked quickly.

(c) Bill ate noisily.

What we will do is treat adverbial phrases as functions over predicates, returning predicates. As such, the type for an adverbial is $(e \rightarrow t) \rightarrow (e \rightarrow t))$. The type tree for such a modification is shown below:



This will be the general strategy in most typed λ-calculi for treating higher order modification relations. An example of the syntactic structure and type tree for a simple binary relation taking proper names is illustrated below.