λ-Y calculus - Definitions

Syntax (nominal)

Types:

$$\sigma ::= a \mid \sigma \to \sigma \text{ where } a \in \mathcal{TV}$$

Terms:

$$M ::= x \mid MM \mid \lambda x.M \mid Y_\sigma \text{ where } x \in Var$$

Well typed terms (nominal)

$$(var)\ \frac{}{\Gamma \vdash x : \sigma}\ (x : \sigma \in \Gamma) \qquad (Y)\ \frac{}{\Gamma \vdash Y_\sigma : (\sigma \to \sigma) \to \sigma}$$

$$(abs)\ \frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (x \sharp \Gamma / x \notin Subjects(\Gamma)) \qquad (app)\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

βY-Reduction(nominal, typed)

$$(red_L)\ \frac{\Gamma \vdash M \Rightarrow M' : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN \Rightarrow M'N : \tau} \qquad (red_R)\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N \Rightarrow N' : \sigma}{\Gamma \vdash MN \Rightarrow M'N : \tau}$$

$$(abs)\ \frac{\Gamma \cup \{x : \sigma\} \vdash M \Rightarrow M' : \tau}{\Gamma \vdash \lambda x.M \Rightarrow \lambda x.M' : \sigma \to \tau}\ (x \sharp \Gamma) \qquad (\beta)\ \frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (\lambda x.M)N \Rightarrow M[N/x] : \tau}\ (x \sharp \Gamma, N)$$

$$(Y)\ \frac{\Gamma \vdash M : \sigma \to \sigma}{\Gamma \vdash Y_\sigma M \Rightarrow M(Y_\sigma M) : \sigma}$$

βY-Reduction(nominal, untyped)

$$(red_L)\ \frac{M \Rightarrow M'}{MN \Rightarrow M'N} \qquad (red_R)\ \frac{N \Rightarrow N'}{MN \Rightarrow M'N}$$

$$(abs)\ \frac{M \Rightarrow M'}{\lambda x.M \Rightarrow \lambda x.M'} \qquad (\beta)\ \frac{}{(\lambda x.M)N \Rightarrow M[N/x]}\ (x \sharp N) \qquad (Y)\ \frac{}{M \Rightarrow M(Y_\sigma M)}$$

Syntax (locally nameless)

Types:

$$\sigma ::= a \mid \sigma \to \sigma \text{ where } a \in \mathcal{TV}$$

Pre-terms:

$$M ::= x \mid n \mid MM \mid \lambda M \mid Y_\sigma \text{ where } x \in Var \text{ and } n \in Nat$$

Open (locally nameless)

$M^N \equiv \{0 \to N\}M$

$\{k \to U\}(x) = x$
$\{k \to U\}(n) = \text{if } k = n \text{ then } U \text{ else } n$
$\{k \to U\}(MN) = (\{k \to U\}M)(\{k \to U\}N)$
$\{k \to U\}(\lambda M) = \lambda(\{k+1 \to U\}M)$
$\{k \to U\}(Y\sigma) = Y\sigma$

Closed terms (locally nameless, cofinite)

$$(fvar)\ \frac{}{\mathrm{term}(x)} \qquad (Y)\ \frac{}{\mathrm{term}(Y_\sigma)}$$

$$(lam)\ \frac{\forall x \notin L.\ \mathrm{term}(M^x)}{\mathrm{term}(\lambda M)}\ (\text{finite } L) \qquad (app)\ \frac{\mathrm{term}(M) \qquad \mathrm{term}(M)}{\mathrm{term}(MN)}$$

βY-Reduction(locally nameless, cofinite, untyped)

$$(red_L)\ \frac{M \Rightarrow M' \qquad \mathrm{term}(N)}{MN \Rightarrow M'N} \qquad (red_R)\ \frac{\mathrm{term}(M) \qquad N \Rightarrow N'}{MN \Rightarrow M'N}$$

$$(abs)\ \frac{\forall x \notin L.\ M^x \Rightarrow M'^x}{\lambda M \Rightarrow \lambda M'}\ (\text{finite } L) \qquad (\beta)\ \frac{\mathrm{term}(\lambda M) \qquad \mathrm{term}(N)}{(\lambda M)N \Rightarrow M^N} \qquad (Y)\ \frac{}{M \Rightarrow M(Y_\sigma M)}$$

Bindings

When describing the (untyped) λ-calculus on paper, the terms of the λ-calculus are usually inductively defined in the following way:

$$t ::= x \mid tt \mid \lambda x.t \text{ where } x \in Var$$

This definition of terms yields an induction/recursion principle, which can be used to define functions over the λ-terms by structural recursion and prove properties about the λ-terms using structural induction (recursion and induction being two sides of the same coin).
Functional languages like Haskell or Isabelle include a way of inductively defining terms and often provide the inductive/recursive principles, such as pattern matching on the term constructors in function definitions or inductive rules based on the definition of terms.

Whilst the definition above describes terms of the lambda calculus, there are implicit assumptions one makes about the terms, namely, the x in the $\lambda x.t$ case appears bound in t. This means that while x and y might be distinct terms of the λ-calculus (i.e. $x \neq y$), $\lambda x.x$ and $\lambda y.y$ represent the same term, as x and y are bound by the λ. Without the notion of α-equivalence of terms, one cannot prove any properties of terms involving bound variables, such as saying that $\lambda x.x \equiv \lambda y.y$.
The solution to this issue is to quotient the terms of the lambda calculus by the relation of α-equivalence. This way, $\lambda x.x \equiv_\alpha \lambda y.y$, because both terms are part of the same equivalence class. In an informal setting, reasoning about lambda terms often involves substitution of one α-equivalent term for another implicitly, to avoid issues with bound names appearing within another term (often referred to as the Barendregt Variable Convention). Indeed, even the usual definition of substitution uses this convention in the lambda case, by implicitly assuming the given lambda term $\lambda y.s$ can always be swapped out for an alpha equivalent term $\lambda y'.s'$, such that y' satisfies the side conditions:

$$(\lambda y'.s')[t/x] \equiv \lambda y'.(s'[t/x]) \text{ assuming } y' \not\equiv x \text{ and } y' \notin FV(t)$$

This definition of substitution is therefore a function on α-equivalence classes of terms, rather than on the "raw" terms. Because of this, the principles of induction/recursion, obtained from the definition of the "raw" terms are no longer sufficient, as this definition is obviously not structurally recursive. In order to reason about the term of the λ-calculus formally, we therefore need a formalization of the terms which provides induction principles for α-equivalent terms.

In general, there are two main approaches taken in a rigorous formalization of the terms of the lambda calculus, namely the concrete approaches and the higher-order approaches, both described in some detail below.

Concrete approaches

The concrete or first-order approaches usually encode variables using names (like strings or natural numbers). Encoding of terms and capture-avoiding substitution must be encoded explicitly. A survey by Aydemir et al. (2008) details three main groups of concrete approaches, found in formalizations of the λ-calculus in the literature:

Named

This approach generally defines terms in much the same way as the informal inductive definition given above. Using a functional language, such as Haskell or ML, such a definition might look like this:

```
datatype trm =
  Var name
| App trm trm
| Lam name trm
```

Since most reasoning about the lambda terms is up to α-equivalence, this notion has to be explicitly stated. There are several ways of doing this, one of which is using nominal sets (described in the section on nominal sets/Isabelle?). The nominal package in Isabelle provides tools to automatically define terms with binders, with notion of alpha equivalence being handled automatically by the package. Using nominal sets in Isabelle results in a definition of terms which looks very similar to the informal presentation of the lambda calculus:

```
nominal_datatype trm =
  Var name
| App trm trm
| Lam x::name l::trm  binds x in l
```

Most importantly, this definition already includes the notion of alpha equivalence, wherein `Lam x (Var x) = Lam y (Var y)` immediately follows. The nominal package also provides freshness lemmas and a strengthened induction principle with name freshness for terms involving binders.

Nameless/de Bruijn

Using a named representation of the lambda calculus in a fully formal setting can be inconvenient when dealing with bound variables. For example, substitution, as described previously, with its side-condition of freshness of y in x and t is not structurally recursive, but rather requires well-founded recursion. To avoid this problem in the definition of substitution, the terms of the lambda calculus can be encoded using de Bruijn indices:

```
datatype trm =
  Var nat
| App trm trm
| Lam trm
```

This representation of terms uses indices instead of named variables. The indices are natural numbers, which encode an occurrence of a variable in a λ-term. For bound variables, the index indicates which λ it refers to, by encoding the number of λ-binders that are in the scope between the index and the λ-binder the variable corresponds to. For example, the term $\lambda x.\lambda y.yx$ will be represented as $\lambda\ \lambda\ 0\ 1$. Here, 0 stands for y, as there are no binders in scope between itself and the λ it corresponds to, and 1 corresponds to x, as there is one λ-binder in scope. To encode free variables, one simply choses an index greater than the number of λ's currently in scope, for example, $\lambda\ 4$.
Since there are no named variables, there is only one way to represent any λ-term, and the notion of α-equivalence is no longer relevant.

To see that this representation of λ-terms is isomorphic to the usual named definition, we can define two function f and g, which translate the named representation to de Bruijn notation and vice versa. More precisely, since we are dealing with α-equivalence classes, its is an isomorphism between these that we can formalize.

To make things easier, we consider a representation of named terms, where we map named variables, $x, y, z, ...$ to indexed variables $x_1, x_2, x_3, ...$ Then, the mapping from named terms to de Bruijn term is given by f, which we define in terms of an auxiliary function e:

$$
\begin{aligned}
e_k^m(x_n) &= \begin{cases} k - m(x_n) - 1 & x_n \in \text{dom } m \\ k + n & otherwise \end{cases} \\
e_k^m(uv) &= e_k^m(u)\ e_k^m(v) \\
e_k^m(\lambda x_n.u) &= \lambda\ e_{k+1}^{m \oplus (x_n, k)}(u)
\end{aligned}
$$

Then $f(t) \equiv e_0^{\emptyset}(t)$

The function e takes two additional parameters, k and m. k keeps track of the scope from the root of the term and m is a map from bound variables to the levels they were bound at. In the variable case, if x_n appears in m, it is a bound variable, and it's index can be calculated by taking the difference between the current index and the index $m(x_k)$, at which the variable was bound. If x_n is not in m, then the variable is encoded by adding the current level k to n.
In the abstraction case, x_n is added to m with the current level k, possibly overshadowing a previous binding of the same variable at a different level (like in $\lambda x_1.(\lambda x_1.x_1)$) and k is incremented, going into the body of the abstraction.

The function g, taking de Bruijn terms to named terms is a little more tricky. We need to replace indices encoding free variables (those that have a value greater than or equal to k, where k is the number of binders in scope) with named variables, such that for every index n, we substitute x_m, where $m = n - k$, without capturing these free variables.

We need two auxiliary functions to define g:

$$
\begin{aligned}
h_k^b(n) &= \begin{cases} x_{n-k} & n \geq k \\ x_{k+b-n-1} & otherwise \end{cases} \\
h_k^b(uv) &= h_k^b(u)\ h_k^b(v) \\
h_k^b(\lambda u) &= \lambda x_{k+b}.\ h_{k+1}^b(u)
\end{aligned}
$$

Im not sure what the sensible notation for these should be...

$$
\begin{aligned}
\Diamond_k(n) &= \begin{cases} n-k & n \geq k \\ 0 & otherwise \end{cases} \\
\Diamond_k(uv) &= \max \Diamond_k(u)\ \Diamond_k(v) \\
\Diamond_k(\lambda u) &= \Diamond_{k+1}(u)
\end{aligned}
$$

The function g is then defined as $g(t) \equiv h_0^{\Diamond_0(t)+1}(t)$. As mentioned above, the complicated definition has to do with avoiding free variable capture. A term like $\lambda(\lambda\ 2)$ intuitively represents a named lambda term with two bound variables and a free variable x_0 according to the definition above. If we started giving the bound variables names in a naive way, starting from x_0, we would end up with a term $\lambda x_0.(\lambda x_1.x_0)$, which is obviously not the term we had in mind, as x_0 is no longer a free variable. To ensure we start naming the bound variables in such a way as to avoid this situation, we use $\Diamond$ to compute the maximal value of any free variable in the given term, and then start naming bound variables with an index one higher than the value returned by $\Diamond$.

As mentioned earlier, checking α-equivalence for de Bruijn notation is the same as checking for syntactic equality, and since the de Bruijn terms are defined inductively, we obtain induction/recursion principles for α-equivalent terms. In their comparison between named vs. nameless/de Bruijn representations of lambda terms, Berghofer and Urban (2006) give details about the definition of substitution, which no longer needs the variable convention and can therefore be defined using primitive structural recursion.
The main disadvantage of using de Bruijn indices is the relative unreadability of both the terms and the formulation of properties about these terms. For example, the substitution lemma, which in the named setting would be stated as:

$$\text{If } x \neq y \text{ and } x \notin FV(L), \text{ then } M[N/x][L/y] \equiv M[L/y][N[L/y]/x].$$

becomes the following statement in the nameless formalization:

$$\text{For all indices } i, j \text{ with } i \leq j,\ M[N/i][L/j] = M[L/j+1][N[L/j-i]/i]$$

Clearly, the first version of this lemma is much more intuitive.

Locally Nameless

The locally nameless approach to minders is a mix of the two previous approaches. Whilst a named representation uses variables for both free and bound variables and the nameless encoding uses de Bruijn indices in both cases as well, a locally nameless encoding distinguishes between the two types of variables. Free variables are represented by names, much like in the named version, and bound variables are encoded using de Bruijn indices. A named term, such as $\lambda x.xy$, would be represented as $\lambda 1y$. The following definition captures the syntax of the locally nameless terms:

```
datatype ptrm =
  Fvar name
  BVar nat
| App trm trm
| Lam trm
```

Note however, that this definition doesn't quite fit the notion of λ-terms, since a `pterm` like `(BVar 1)` does not represent a λ-term, since bound variables can only appear in the context of a lambda, such as in `(Lam (BVar 1))`.

Higher-Order approaches

Unlike concrete approaches to formalizing the lambda calculus, where the notion of binding and substitution is defined explicitly in the host language, higher-order formalizations use the function space of the implementation language, which handles binding. This way, the definitions of capture avoiding substitution or notion of α-equivalence are offloaded onto the meta-language.

HOAS

HOAS, or higher-order abstract syntax (F. Pfenning and Elliott 1988, Harper, Honsell, and Plotkin (1993)), is a framework for defining logics based on the simply typed lambda calculus. A form of HOAS, introduced by Harper, Honsell, and Plotkin (1993), called the Logical Framework (LF) has been implemented as Twelf by Frank Pfenning and Schürmann (1999).
Using HOAS for encoding the λ-calculus comes down to encoding binders using the meta-language binders. An example from the Polakow (2015) paper on embedding linear lambda calculus in Haskell encodes lambda terms as:

```
data Exp a where
  Lam :: Exp a -> Exp b -> Exp (a -> b)
  App :: Exp (a -> b) -> Exp a -> Exp b
```

This definition avoids the need for explicitly defining substitution, because it uses Haskell's variables, which already include the necessary definitions. However, using HOAS only works if the notion of α-equivalence and substitution of the meta-language coincide with these notions in the object-language.

Weak Higher-Order?

References

Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. "Engineering Formal Metatheory." In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 3–15. POPL '08. New York, NY, USA: ACM. doi:10.1145/1328438.1328443.

Berghofer, Stefan, and Christian Urban. 2006. "A Head-to-Head Comparison of de Bruijn Indices and Names." In *IN PROC. INT. WORKSHOP oN LOGICAL FRAMEWORKS aND METALANGUAGES:*

THEORY aND PRACTICE, 46–59.

Harper, Robert, Furio Honsell, and Gordon Plotkin. 1993. "A Framework for Defining Logics." *J. ACM* 40 (1). New York, NY, USA: ACM: 143–84. doi:10.1145/138027.138060.

Pfenning, F., and C. Elliott. 1988. "Higher-Order Abstract Syntax." In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, 199–208. PLDI '88. New York, NY, USA: ACM. doi:10.1145/53990.54010.

Pfenning, Frank, and Carsten Schürmann. 1999. "Automated Deduction — CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings." In, 202–6. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-48660-7_14.

Polakow, Jeff. 2015. "Embedding a Full Linear Lambda Calculus in Haskell." In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 177–88. Haskell '15. New York, NY, USA: ACM. doi:10.1145/2804302.2804309.