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)

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

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

βY-Reduction(locally nameless, cofinite, untyped)

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

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

Bindings

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

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

Whilst the case of variables (x) and application (tt) is fairly straight forward, appearing identically in a more formal/rigorous setting of a theorem prover or a programming language, there are assumptions we implicitly make about the lambda case ($\lambda x.t$). Specifically, the lambda case introduces a variable x, which may appear bound within the body t of the lambda expression. The consequence of this binding in an informal setting includes implicitly adopting a lambda equivalence on terms, where $\lambda x.x$ and $\lambda y.y$ intuitively represent the same lambda term. In this informal setting, reasoning about lambda terms often involves the use of the Barendregt Variable Convention to pick out suitably fresh bound variables in a term, intuitively relying on the α-equivalence of lambda terms to swap out one such term for another. Indeed, even the usual definition of substitution assumes this convention in the lambda case, namely 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:

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

Whilst this approach allows for much simpler and more elegant proofs on paper, in a formal setting of a theorem prover, the notions of alpha equivalence terms and the use of Barendregt Variable Convention have to be formalized and used rigorously. 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 natural numbers instead of named variables. As a result, the notion of α-equivalence is no longer relevant, as all terms encoded this way are invariant under α-conversion. As an example, both $\lambda x.x$ and $\lambda y.y$ are written as $\lambda 1$, using de Brujin indices. The natural number denotes the number of lambda's that are in scope between the variable and it's corresponding lambda. In their comparison between named vs. nameless/de Bruijn representations of lambda terms, Berghofer and Urban (2006) give further 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 this approach 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 idices 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 something like `(BVar 1)` does not represent a term (since bound variables can only appear in the context of a lmbda, such as `(Lam (BVar 1))`).

Higher-Order approaches

HOAS

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.