A formalization of the λ-Y calculus

Samuel Balco

Abstract

This is the abstract. For this and the other front-matter options you can either include the text directly on the metadata file or you can use in order to include your text.

Contents

Introduction

Formal verification of software is a field of active research in computer science. One of the main approaches to verification is model checking, wherein a system specification is checked against certain correctness properties, by finding a model of the system, encoding the desired correctness property as a logical formula and then exhaustively checking whether the given formula is satisfiable in the model of the system. Big advances in model checking of 1^{st} order (imperative) programs have been made, with techniques like abstraction refinement and SAT/SMT-solver use, allowing scalability.
Higher order (functional) program verification, on the other hand, has been much less explored. Current approaches to formal verification of such programs usually involve the use of (automatic) theorem provers, which usually require a lot of user interaction and as a result have not managed to scale as well as model checking in the 1^{st} order setting. In recent years, advances in higher order model checking (HOMC) have been made by Ong / ? (find paper??). Whilst a lot of theory has been developed for HOMC, there has been little done in implementing/mechanizing these results in a fully formal setting of a theorem prover.
The aim of this project is to make a start of such a mechanization, by formalizing the λ-Y calculus with the intersection-type system described by ? and formally proving important properties of the system.
The first part of this work focuses on the mechanization aspect of the simply typed λ-Y calculus in a theorem prover, in a fashion similar to the PoplMark challenge, by exploring different formalizations of the calculus and the use of different theorem provers. The project focuses on the engineering choices and formalization overheads, which result from translating the informal systems into a fully-formal setting of a theorem prover.

Binders

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).
However, whilst the definition above describes valid terms of the λ-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 $\lambda x.x \equiv \lambda y.y$.

In an informal setting, reasoning with α-equivalence of terms is often very implicit, however in a formal setting of theorem provers, having an inductive definition of "raw" *lambda*-terms, which are not *alpha*-equivalent, yet reasoning about α-equivalent λ-terms

poses certain challenges.
One of the main problems is the fact that the inductive/recursive definition does not easily lift to *alpha*-equivalent terms. Take a trivial example of a function on raw terms, which checks whether a variable appears bound in a given λ-term. Clearly, such function is well formed for "raw" terms, but does not work (or even make sense) for α-equivalent terms.
Conversely, there are informal definitions over α-equivalent terms, which are not straight-forward to define over raw terms. Take the usual definition of substitution, defined over α-equivalent terms, which actually relies on this fact in the following case:

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

Here in the λ case, it is assumed that a 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 condition. The assumption that a bound variable can be swapped out for a "fresh" one to avoid name clashes is often referred to as the Barendregt Variable Convention.

The direct approach of defining "raw" terms and an additional notion of α-equivalence introduces a lot of overhead when defining functions, as one either has to use the recursive principles for "raw" terms and then show that the function lifts to the α-equivalent terms or define functions on *alpha*-equivalence classes and prove that it is well-founded, without being able to rely on the structurally inductive principles that one gets "for free" with the "raw" terms.
Because of this, the usual informal representation of the λ-calculus is rarely used in a fully formal setting.

To mitigate the overheads of a fully formal definition of the λ-calculus, we want to have an encoding of the λ-terms, which includes the notion of α-equivalence whilst being inductively defined, giving us the inductive/recursive principles for *alpha*-equivalent terms directly. This can be achieved in several different ways. 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
```

As was mentioned before, defining "raw" terms and the notion of α-equivalence of "raw" terms separately carries a lot of overhead in a theorem prover and is therefore not favored.

To obtain an inductive definition of λ-terms with a built in notion of α-equivalence, one can instead use nominal sets (described in the section on nominal sets/Isabelle?). The nominal package in Isabelle provides tools to automatically define terms with binders, which generate inductive definitions of α-equivalent terms. 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 allows one to define functions over α-equivalent terms using structural induction. 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 in the introduction, with its side-condition of freshness of y in x and t is not structurally recursive on "raw" terms, but rather requires well-founded recursion over α-equivalence classes of terms. 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, $\lambda\ 4$.

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, $x, y, z, \ldots$ to indexed variables $x_1, x_2, x_3, \ldots$. Then, the mapping from named terms to de Bruijn term is given by f, which we define in terms of an auxiliary function e:

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

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:

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

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

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 one quickly notices, a term like $\lambda x.x$ and $\lambda y.y$ have a single unique representation as a *deBruijnterm* $\lambda\ 0$. Indeed, since there are no named variables in a de Bruijn term, there is only one way to represent any λ-term, and the notion of α-equivalence is no longer relevant. We thus get around our problem of having an inductive principle and α-equivalent terms, by having a representation of λ-terms where every α-equivalence class of λ-terms has a single representative term in the de Bruijn notation.

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 binders 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. By using de Bruijn indices for bound variables, we again obtain an inductive definition of terms which are already *alpha*-equivalent.

While closed terms, like $\lambda x.x$ and $\lambda y.y$ are represented as de Bruijn terms, the term $\lambda x.xz$ and $\lambda x.xz$ are encoded as $\lambda\ 0z$. 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))`.

The advantage of using a locally nameless definition of λ-terms is a better readability of such terms, compared to equivalent de Bruijn terms. Another advantage is the fact that definitions of functions and reasoning about properties of these terms is much closer to the informal setting.

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. 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), which has been previously used to encode the λ-calculus.
Using HOAS for encoding the λ-calculus comes down to encoding binders using the meta-language binders. This way, the definitions of capture avoiding substitution or notion of α-equivalence are offloaded onto the meta-language. As an example, take the following definition of terms of the λ-calculus in Haskell:

```
data Term where
  Var :: Int -> Term
  App :: Term -> Term -> Term
  Lam :: (Term -> Term) -> Term
```

This definition avoids the need for explicitly defining substitution, because it encodes a lambda term as a Haskell function `(Term -> Term)`, relying on Haskell's internal substitution and notion of α-equivalence. As with the de Bruijn and locally nameless representations, this encoding gives us inductively defined terms with a built in notion of α-equivalence.
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.

Methodology

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)

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

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

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[1].

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[2].

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[3].

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[4].

[1] https://doi.org/10.1145/1328438.1328443
[2] https://doi.org/10.1145/138027.138060
[3] https://doi.org/10.1145/53990.54010
[4] https://doi.org/10.1007/3-540-48660-7_14