

Formally Building the Theory of the λ -Calculus

Naman Bharadwaj

May 9, 2015

Abstract

For this project, I explore the use of an interactive theorem prover to build the foundational theory of the simply typed lambda calculus with natural numbers. After encoding the simple syntax and type system for the language, I describe small-step operational semantics, formally define the complex notions of α -equivalence and substitution, and define the axiomatic (equational) theory of α - β - η -equivalence.

1 Introduction

My motivation for this project is to develop a better understanding of the power of modern theorem proving and checking tools in the area of verifying correctness properties of programs. Next year, I will be working at a financial technology startup helping to ensure the correctness of critical currency-handling programs. I hope to be able to use some of the techniques and insights I picked up through my work on this project.

For this project, I picked up Lean, a new theorem prover being developed at Microsoft Research and Carnegie Mellon University. It is based on the calculus of inductive constructions, and is, in many respects, similar to Coq. Due to the ongoing development of both the language and the libraries, the use of Lean presented challenges involving lack of language features and succinct, comprehensive documentation (in fact, most of my debugging involved searching through the Lean source code).

My initial plan for this project was to continue to add polymorphic abstraction to the language, and verify some simple but concrete programs (such as basic natural number arithmetic). The development of the foundational theory, however, turned out to be a complex task, especially when learning a new language concurrently. Thus, I simply completed as much of the foundational theory as possible before the project deadline.

I will now provide an outline of my experience implementing the theory in Lean. I will not go into excessive detail, as I also provide the code that implements what I discuss.

2 The Language, Syntax, and Types

The language that I ultimately formalized is just the simply typed lambda calculus with natural numbers:

$$\begin{aligned} \tau &:= \text{nat} \mid \tau_1 \rightarrow \tau_2 \\ c &:= z \mid s \\ e &:= c \mid x \mid \lambda x : \tau. e \mid e_1 e_2 \end{aligned}$$

$$\begin{array}{c} \frac{}{\Gamma \vdash z : \text{nat}} \qquad \frac{}{\Gamma \vdash s : \text{nat} \rightarrow \text{nat}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\ \\ \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2} \end{array}$$

The syntax and typing judgement specified above are simple enough to formulate as an inductive type and proposition, respectively. We have yet to clarify what we mean by Γ , however.

2.1 Typing Contexts

Γ in the typing judgement $\Gamma \vdash e : \tau$ is a typing context – just a map from variables to types. So in order to formalize Γ we need to develop tables and some theory around them (due to the sparseness of the Lean standard library). Initially, I attempted to encode tables as binary search trees, but quickly abandoned that approach due to unnecessary complexity.

Tables mapping values of type τ_1 to values of type τ_2 in my implementation, are simply functions of type $\tau_1 \rightarrow \text{option } \tau_2$. Then the judgement above, $x : \tau \in \Gamma$, is just the proposition $\Gamma x = \text{option.some } \tau$. In order to be complete, we must also make sense of $\Gamma, x : \tau_1$. What we really mean here is the (right-biased) union of the typing context Γ with the singleton context $x : \tau_1$.

Table union is just defined as $\Gamma_1 \cup \Gamma_2 = \Gamma_3$ iff $x : \tau \in \Gamma_3 \leftrightarrow x : \tau \in \Gamma_2 \vee (x : \tau \in \Gamma_1 \wedge x \notin \Gamma_2)$. Finally, we must show that we can always take the union of two tables and get back a third; this is done simply by defining a union function and proving that it is correct w.r.t. the definition of union above.

2.2 Proofs about Types

Next, we can go ahead and formalize some simple proofs about our typing judgement. First, we may state simple inversion lemmas that allow us to say something about the structure of τ by looking at the structure of e , assuming $\Gamma \vdash e : \tau$.

The first non-trivial proof is *unicity of typing*, or injectivity of the typing judgement. This theorem states that if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$.

This proof is straightforward induction over the proofs of the typing judgements; in each case we must apply the appropriate inversion lemmas.

In one case, however, we need the extra lemma that if $\Gamma_1 \cup \Gamma_2 = \Gamma_3$ and $\Gamma_1 \cup \Gamma_2 = \Gamma_4$, then $\Gamma_3 = \Gamma_4$. Unfortunately, this is not true from an intensional view of function equality (where the algorithm matters, not just the behavior). One solution here is to define a new equivalence relation on tables and then prove that you can substitute equivalent tables for each other in certain circumstances. In order to avoid this complexity, I opted to just use the function extensionality axiom, which provides a view of functions where $(\forall x. fx = gx) \rightarrow f = g$.

2.3 Typechecking

Finally, we define a *typechecking* function that takes a context Γ and expression e and outputs a type τ such that $\Gamma \vdash e : \tau$. There are two non-trivial proofs here; the easier of the two is soundness – if the typechecker outputs τ , then in fact, $\Gamma \vdash e : \tau$. The harder one is completeness – if $\Gamma \vdash e : \tau$, then the typechecker will find and output τ . Completeness uses our function extensionality explained above.

3 Substitution

In order to continue on to semantics, we must now stop and consider the definition of capture-avoiding substitution. I initially ignored this, in the hopes that all substituted expressions would be closed terms, but open term substitution is necessary in order to specify an axiomatic semantics.

Substitution is a complex idea, and depends on the idea of α -equivalence. Intuitively, α -equivalence captures the idea that it doesn't matter what we call a bound variable, as long as all the references to that variable are valid. Unfortunately, α -equivalence itself relies on the ability to replace free occurrences of a variable with another variable, while avoiding variable capture. Thus I define substitution and α -equivalence mutually inductively, with the extra judgement $x \in FV(e)$ asserting that x appears as a free variable in the expression e :

$$\begin{array}{c}
\frac{}{e =_{\alpha} e} \qquad \frac{e_1 =_{\alpha} e'_1 \quad e_2 =_{\alpha} e'_2}{e_1 e_2 =_{\alpha} e'_1 e'_2} \qquad \frac{e_1 =_{\alpha} e_2 \quad e_2 \rightsquigarrow_x^y e_3}{\lambda x : \tau. e_1 =_{\alpha} \lambda y : \tau. e_3} \\
\\
\frac{}{x \rightsquigarrow_x^e e} \qquad \frac{}{y \rightsquigarrow_x^e y} \qquad \frac{}{c \rightsquigarrow_x^e c} \qquad \frac{e_1 \rightsquigarrow_x^e e'_1 \quad e_2 \rightsquigarrow_x^e e'_2}{e_1 e_2 \rightsquigarrow_x^e e'_1 e'_2} \\
\\
\frac{z \notin FV(e) \quad \lambda y : \tau. e_1 =_{\alpha} \lambda z : \tau. e'_1 \quad e'_1 \rightsquigarrow_x^e e''_1}{\lambda y : \tau. e_1 \rightsquigarrow_x^e \lambda z : \tau. e''_1} \\
\\
\frac{}{x \in FV(x)} \qquad \frac{x \in FV(e)}{x \in FV(\lambda y : \tau. e)} \qquad \frac{x \in FV(e_1)}{x \in FV(e_1 e_2)} \qquad \frac{x \in FV(e_2)}{x \in FV(e_1 e_2)}
\end{array}$$

Note that the substitution judgement is not well-moded. That is, there is not only one possible substitution of e for x in e_1 . This is because each α -equivalence class is infinite in size! In particular, finding an α -conversion in order to perform a substitution is not a deterministic task.

There are three main theorems of interest, one which can only be proven after defining our semantics below. First, we want that for every e_1, x, e , we there exists an e'_1 such that $e_1 \rightsquigarrow_x^e e'_1$. This involves implementing a substitution algorithm, which is ultimately very tricky and slow. We state this as an axiom in order to continue to more interesting things. The second is the substitution lemma, which states if $\Gamma, x : \tau \vdash e_1 : \tau_1, \Gamma \vdash e : \tau$, and $e_1 \rightsquigarrow_x^e e'_1$, then $\Gamma \vdash e'_1 : \tau_1$. Finally, we would like to define our operational semantics such that it preserves the $=_\alpha$ relation.

4 Semantics

4.1 Operational

The small-step operational semantics I define are the standard semantics for the eagerly-evaluated lambda calculus:

$$\begin{array}{c} \frac{}{c \text{ val}} \qquad \frac{}{\lambda x : \tau. e \text{ val}} \qquad \frac{e \text{ val}}{c e \text{ val}} \\ \\ \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} \qquad \frac{e_1 \text{ val} \quad e_2 \rightarrow e'_2}{e_1 e_2 \rightarrow e_1 e'_2} \qquad \frac{e_1 \rightsquigarrow_x^e e'_1}{(\lambda x : \tau. e_1) e \rightarrow e'_1} \end{array}$$

The proofs of interest here are progress – if $\emptyset \vdash e : \tau$, then either $e \text{ val}$ or $\exists e'. e \rightarrow e'$ – and preservation, which simply states that the typing judgement is preserved by the stepping judgement. We can also now prove the theorem described in the previous section, that α -equivalent expressions step to α -equivalent expressions.

4.2 Denotational

We would like to interpret (possibly open) expressions as mathematical objects. My approach is to instead interpret typed expressions (actually, the typing judgement). First, we interpret types as simply natural numbers and functions:

$$\begin{aligned} \llbracket \text{nat} \rrbracket &= \mathbb{N} \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \{f \mid \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\} \end{aligned}$$

Next, in order to interpret open terms, we define the concept of a Γ -environment, a member of $\llbracket \Gamma \rrbracket$.

$$\llbracket \Gamma \rrbracket = \{f \mid x : \tau \in \Gamma \rightarrow f(x) \in \llbracket \tau \rrbracket\}$$

Next, we interpret typing judgements $\Gamma \vdash e : \tau$ as functions of type $\llbracket \Gamma \rrbracket \rightarrow \llbracket \tau \rrbracket$.

$$\begin{aligned} \llbracket \Gamma \vdash z : \mathbf{nat} \rrbracket(f) &= 0 \\ \llbracket \Gamma \vdash s : \mathbf{nat} \rightarrow \mathbf{nat} \rrbracket(f)(x) &= x + 1 \\ \llbracket \Gamma \vdash x : \tau \rrbracket(f) &= f(x) \\ \llbracket \Gamma \vdash e_1 e_2 : \tau_2 \rrbracket(f) &= \llbracket \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \rrbracket(f)(\llbracket \Gamma \vdash e_2 : \tau_1 \rrbracket(f)) \\ \llbracket \Gamma \vdash \lambda x : \tau_1. e : \tau_1 \rightarrow \tau_2 \rrbracket(f)(v) &= \llbracket \Gamma, x : \tau_1 \vdash e : \tau_2 \rrbracket(f \cup (x \mapsto v)) \end{aligned}$$

We can now prove that if $\Gamma \vdash e : \tau$ and $e \rightarrow e'$, then $\llbracket \Gamma \vdash e : \tau \rrbracket = \llbracket \Gamma \vdash e' : \tau \rrbracket$. In other words, the denotational semantics are preserved by the small-step operational semantics described previously.

4.3 Axiomatic

The final step is to introduce the notion of β - η -equivalence, which is our extensional equational theory:

$$\frac{}{e =_{\beta} e} \quad \frac{e_1 \rightsquigarrow_x^e e'_1}{(\lambda x : \tau. e_1)e =_{\beta} e'_1} \quad \frac{e_1 =_{\beta} e'_1 \quad e_2 =_{\beta} e'_2}{e_1 e_2 =_{\beta} e'_1 e'_2} \quad \frac{x \notin FV(e)}{e =_{\eta} \lambda x : \tau. e x}$$

Along with α -equivalence, we now should have a complete equational theory of the simply typed lambda calculus with natural numbers. We will let $e \equiv e'$ mean that e and e' are α - β - η equivalent. Finally, we should show that $\llbracket \Gamma \vdash e : \tau \rrbracket = \llbracket \Gamma \vdash e' : \tau \rrbracket$ iff $e \equiv e'$ (that this is, in fact, a sound and complete equational theory).

5 Next Steps

Unfortunately, I did not have time to write as much code as I had originally planned for (some of the above is not 100% formalized in Lean). I now outline some ideas for further exploration, that I hope to be able to do:

Verify Some Programs I hope to continue on to do this even after submission of the project, since the foundations are mostly laid. I would like to implement standard arithmetic functions, such as addition, multiplication, etc. and prove them correct with respect to the denotational semantics.

Polymorphic Abstraction My initial intent was to develop the full theory of System F, including polymorphic abstraction. This makes all the theorems that I have proven reasonably more complex, and is thus a non-trivial amount of work.

Categorical Semantics Of particular interest to me is to formalize the simply typed lambda calculus as the internal language of cartesian closed categories. The standard method (at least, the one that I have seen) of doing so involves adding explicit product types, $\tau_1 \times \tau_2$, to the language. An alternative approach could be to introduce polymorphism as above, and then encode $\tau_1 \times \tau_2$ as $\forall \alpha. (\tau_1 \rightarrow \tau_2 \rightarrow \alpha) \rightarrow \alpha$. However, it is not entirely clear what the implications of adding full polymorphism to the language, as the semantics of polymorphism in category theory is unclear (and this would thus require a good amount of theoretical work).

IPC Another interesting angle is to develop the full theory of the intuitionistic propositional calculus by adding binary sums, binary products, and bottom to the language. We could then proceed to formalize various semantics, such as the Heyting algebra semantics or Kripke semantics.