

CS256: Programming Languages and Semantics

Simply-Typed Lambda Calculus

<http://www.eecs.harvard.edu/~greg/cs256sp2006/>

Greg Morrisett

greg@eecs.harvard.edu

327 Maxwell Dworkin

February 22, 2006

1 Overview

At this point, you've seen simple instances of the three major kinds of models that we build for programming languages: operational, denotational, and axiomatic. This was all done in the context of IMP to keep things relatively simple and our attention on how the models are built (and used) instead of the language. Now we're going to shift gears and start looking at linguistic features, how to model them, and their interactions. (I recommend picking up Pierce's book at this point for background reading.)

We're going to start with procedures (functions) in the context of the simply-typed lambda calculus. Functions introduce a number of key concepts, including variables, scope, substitution, re-naming, encapsulation, etc. We're going to study a number of different operational models for the lambda calculus, as well as some denotational models. (Good axiomatic models are still an open issue as far as I'm concerned, so we'll give them shorter shrift until the end of the course.)

Then we'll start growing the language to encompass other standard features including control mechanisms (exceptions, continuations, threads, etc.), and dynamic state (i.e., references). Our treatment of these features will revolve around types for a number of reasons: First, types provide both a logical and mathematical "skeleton" on which to hang the clothes of the language. Second, types themselves introduce a number of linguistic mechanisms, such as subtyping, polymorphism, and inductive and recursive types. Third, the structure provided by types is invaluable for building certain classes of proofs.

As an aside, you might wonder why we start with the simply-typed lambda calculus instead of the untyped lambda calculus. The reason is that, while both have very simple operational semantics, the denotational semantics of the "untyped" lambda calculus forces us to assign it some notion of type, and that notion is relatively complicated compared to simple types. In particular, we need some form of recursive types which demand more care to treat formally. In short, the simply-typed lambda calculus is simpler!

2 Syntax

The BNF for the simply-typed lambda calculus is given below:

$$\begin{array}{l} \text{(types)} \quad \tau ::= b \mid \tau_1 \rightarrow \tau_2 \\ \text{(terms)} \quad e ::= c \mid x \mid \lambda x:\tau.e \mid e_1 e_2 \end{array}$$

In the definitions above, x ranges over a denumerable class of identifiers, b is some base type (e.g., int) and c ranges over some constants of that type (e.g., 42).

Terms of the form $\lambda x:\tau.e$ are anonymous functions with a formal parameter names x of type τ , and a body e that typically makes use of x .

Terms of the form $e_1 e_2$ are applications or function calls. The intention is that e_1 should evaluate to a λ -term, and we're going to pass e_2 to that function as an actual parameter. By convention, applications are written as left-associative. So for instance, $e_1 e_2 e_3$ should be parsed as $(e_1 e_2) e_3$.

This raises a big point: I write out terms like " $\lambda x : b. \lambda y : b \rightarrow b.yx$ " to represent a particular lambda term and we use parsing conventions to (unambiguously) determine what term the string is meant to represent. (We were doing the same thing for IMP expressions and commands.) That is, when writing down the models of the language, the terms we manipulate have more structure than strings of symbols like "x" and " λ ", and we can in fact consider these objects to be structured trees which are isomorphic to fully parenthesized terms. In other words, by convention, we treat " $(e_1 e_2) e_3$ ", " $e_1 e_2 e_3$ " and " $((e_1) (e_2)) (e_3)$ " as being structurally or syntactically equivalent because they all parse into the same tree data structure. This is the essence of *abstract syntax* as opposed to *concrete syntax*. We're manipulating trees, not strings.

Now for the lambda-calculus, and indeed any language that has a notion of lexically-scoped variables, it's convenient to go a step further and talk about manipulating *graphs* instead of just trees. The reason is that when we write " $\lambda x. \lambda f. (f x)$ ", the choice of variable names is not meant to matter. If we could write out the term as a graph, then the free occurrence of f in the body of the function would point back to the inner-most lambda which defines the variable, while the free occurrence of x would point back to the outer-most lambda. We've introduced names for those lambdas just so that we can write out the term as a string or tree. This then, is the principle that lexically-scoped variables don't really have names, but are rather, artifacts of the notation we use to express terms.

An alternative, variable-less syntax for terms is as follows and due to DeBruijn:

$$\begin{array}{l} \text{(nats)} \quad n ::= 0 \mid \text{succ}(n) \\ \text{(terms)} \quad e ::= c \mid \underline{n} \mid \lambda \tau. e \mid e_1 e_2 \end{array}$$

Here, we use natural numbers in lieu of free variables. The natural number $\underline{0}$ refers to the inner most λ . In general, \underline{n} refers to the n^{th} enclosing lambda, working our way back towards the root of the term tree. For instance, $\lambda x. \lambda f. (f x)$ is represented as $\lambda. \lambda. (\underline{0} \underline{1})$ where I use "1" as short-hand for $\text{succ}(0)$.

This encoding makes it particularly easy to see the graph structure that we're trying to represent with strings or trees: all of the naming is relative, just like pronouns in natural language. The other thing that's nice about the DeBruijn representation is that it's clear that two terms which have the same graph are equal iff they have the same syntactic form as DeBruijn terms. This is not so with the standard, variable-based treatment.

On the other hand, reading DeBruijn terms of any complexity is a pain in the ass. Humans like using names for place holders and of course, picking good names for variables is crucial for understanding their intended role. Additionally, the notion of substitution for DeBruijn terms is complicated by the fact that we must adjust offsets as we substitute actual arguments for their corresponding index. Finally, DeBruijn notation can only talk about *closed* terms, whereas occasionally, we're going to need to talk about *open* terms (terms with free variables.)

To rectify this situation, I'll be using named λ -terms. However, it's important to note that those named terms will actually represent the underlying graphs (or equivalently, their DeBruijn trees).

2.1 Alpha Equivalence and Substitution

We can formalize these ideas by defining a notion of equivalence on terms called α -equivalence and written $e_1 \equiv_{\alpha} e_2$. The idea is that α -equivalent terms should have the same graph or DeBruijn terms. Furthermore, when we start writing down the operational semantics or typing rules for the language, we'll do so with the understanding that they are really relations on the α -equivalence classes. For instance, if I write " $\lambda x. e_1$ " in a rule, then this is short-hand for $[\lambda x. e_1]_{\alpha}$.

To formalize this, we must first define a few notions: The set of *free variables* of a term ($FV(e)$) is defined

by:

$$\begin{aligned}
FV(c) &= \emptyset \\
FV(x) &= \{x\} \\
FV(\lambda x:\tau.e) &= FV(e) \setminus \{x\} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}$$

The set of *bound variables* of a term ($BV(e)$) is defined by:

$$\begin{aligned}
FV(c) &= \emptyset \\
BV(x) &= \emptyset \\
BV(\lambda x:\tau.e) &= \{x\} \cup BV(e) \\
BV(e_1 e_2) &= BV(e_1) \cup BV(e_2)
\end{aligned}$$

We define (possibly capturing) substitution of the term e_2 for the variable x in the term e_1 (written $e_1\{e_2/x\}$) as follows:

$$\begin{aligned}
c\{e_2/x\} &= c \\
x\{e_2/x\} &= e_2 \\
y\{e_2/x\} &= y && (y \neq x) \\
(e e')\{e_2/x\} &= (e\{e_2/x\})(e'\{e_2/x\}) \\
(\lambda x:\tau.e)\{e_2/x\} &= \lambda x:\tau.e \\
(\lambda y:\tau.e)\{e_2/x\} &= \lambda y:\tau.(e\{e_2/x\}) && (y \neq x)
\end{aligned}$$

We define α -equivalence as the smallest equivalence relation (i.e., includes reflexivity, transitivity, and symmetry) that respects the following rules

$$\frac{e_1 \equiv_\alpha e'_1 \quad e_2 \equiv_\alpha e'_2}{e_1 e_2 \equiv_\alpha e'_1 e'_2}$$

$$\frac{e \equiv_\alpha e'[x/y] \quad x \notin BV(e')}{\lambda x:\tau.e \equiv_\alpha \lambda y:\tau.e'}$$

One way to use the second rule is to just pick a term with fresh variables everywhere, and show it's α -equivalent to both terms. We're guaranteed that such a term exists, because there are a finite number of lambdas in any term, and a countably infinite number of variables in the language.

Finally, we define $[e]_\alpha$ to be:

$$\{e' \mid e' \equiv_\alpha e\}$$

All of this is very tedious of course, so from henceforth, I will treat all terms modulo α -equivalence. It's important to note that this rarely bites us, except, of course, when we go to mechanize things. More on this later...

3 Large Step, Dynamic Semantics

Next, we want to formalize the evaluation of terms in the language. I'll start with a big-step semantics. Configurations consist of just terms (there's no state component yet) and the "good" final configurations are called *values* and consist of either constants (c) or functions ($\lambda x:\tau.e$).

$$(\text{values}) \quad v ::= c \mid \lambda x:\tau.e$$

There are only two rules for the language. The first says that values evaluate to themselves:

$$v \Downarrow v$$

The second describes the evaluation of applications:

$$(\beta) \quad \frac{e_1 \Downarrow \lambda x:\tau.e \quad e_1[e_2/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

This rule uses the notion of *capture-avoiding* substitution ($e_1[e_2/x]$) which is similar to the definition above but subtly and importantly different:

$$\begin{aligned}
c[e_2/x] &= c \\
x[e_2/x] &= e_2 \\
y[e_2/x] &= y && (y \neq x) \\
(e e')[e_2/x] &= (e[e_2/x]) (e'[e_2/x]) \\
(\lambda x:\tau.e)[e_2/x] &= \lambda x:\tau.e \\
(\lambda y:\tau.e)[e_2/x] &= \lambda y:\tau.(e[e_2/x]) && (y \neq x \text{ and } y \notin FV(e_2))
\end{aligned}$$

Notice that we can only substitute under a λ when there's no danger that we'll accidentally capture a free variable in e_2 . Although this seems to make substitution a partial function, the fact that we're really manipulating α -equivalence classes ensures that we can pick a representative such that there's no conflict.

Notice that in the β -rule, I've chosen a call-by-name (CBN) strategy here where we don't evaluate the argument before substituting it for the formal parameter. An alternative is the call-by-value strategy (CBV) where we first evaluate the argument to a value before performing the substitution:

$$(\beta v) \frac{e_1 \Downarrow \lambda x:\tau.e \quad e_2 \Downarrow v' \quad e_1[v'/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

The intention here is that we're modeling the functional aspects of languages such as Haskell or ML. Functions are first-class objects that can be passed into or out of other functions. The lexical scoping of variables is reflected in the fact that our model essentially treats names as irrelevant.

A small-step semantics is also straightforward to formalize. For CBN, the rules might look like this:

$$\begin{aligned}
(\lambda x:\tau.e_1) e_2 &\mapsto e_1[e_2/x] \\
\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}
\end{aligned}$$

For CBV, the rules are slightly different:

$$\begin{aligned}
(\lambda x:\tau.e_1) v &\mapsto e_1[v/x] \\
\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \\
\frac{e_2 \mapsto e'_2}{v e_2 \mapsto v e'_2}
\end{aligned}$$

Note how the rules force a left-to-right, inner-most to outer-most evaluation order.

What are the tradeoffs for CBN and CBV? In principle, CBN can avoid doing some unnecessary computations. In particular, for $(\lambda x.e_1) e_2$, if x is never used in e_1 , then we shouldn't bother to evaluate e_2 . However, such situations arise rarely, and of course, if x is used more than once, a CBN strategy forces us to re-do the work. Hence, CBV is used in practice for most languages (e.g., ML, Java, Scheme, C, etc.) An alternative strategy is call-by-need, where we evaluate each term at most once. On paper, it seems to combine the best of both strategies, which is one of the reasons it's used by languages such as Haskell or Clean. However, (and this is *my* opinion) there are compelling semantic and implementation reasons why it is not so appealing. For one thing, it demands a lot more machinery to model accurately.

One other issue is worth mentioning: these models are good for capturing input/output relationships, but not so good at capturing the actual costs of evaluation (even in a big-O sense) because we're modeling substitution as an atomic transition. We'll talk about lower-level models that deal with this particular issue later....

4 Static Semantics

Unlike IMP, there are terminal (or stuck) configurations for our lambda calculus terms that are not “good” (i.e., values). In particular, if we encounter a free variable x , or if we try to apply a constant as if it were a function (e.g., ce) then no rule applies.

The goal of a type system (or static semantics) is to cut down the language of terms so that we only have expressions that won’t become stuck when evaluated. There are two ways to achieve this goal: You could just add more rules to the operational semantics so that the \mapsto relation is defined on all (non-value) configurations. For instance, we could add the rules:

$$x \mapsto c$$

and

$$ce \mapsto e$$

or perhaps add a new constant, corresponding to an “error value” that we use to record that an error occurred. This is the approach taken by languages such as Scheme. There are at least two disadvantages to this approach to achieving type-safety: First, we may unexpectedly change the desired algebraic structure of operators in the language. For instance, if we say that any value divided by zero (usually a stuck term) steps to the value 3, then we will invalidate certain arithmetic identities that we wish to use in reasoning about programs. Second, an implementation of the language is forced to discriminate at run-time to determine what rule to apply. For instance, if we allow constants to be “applied” to arguments, then at run-time, we must check to see whether the “function” is indeed a function or a constant to determine which rule to invoke. In turn, this means that we must retain enough information (e.g., type-tags) on values to be able to perform the necessary checks and dispatch to the correct code.

4.1 An Aside

Languages such as C and C++ take a different approach: They allow implementors to transition configurations which are “undefined” to an arbitrary *concrete* machine state. The problem is that such machine states may be outside of the space of our *abstract* machine configurations. Thus, for C and C++, we cannot even say what possible states an ill-defined expression might step to. So the point of “type-safety” isn’t really about types at all, but rather, is a statement that the abstract model is a model that indeed captures all (relevant) aspects of the language and hence, we can reason at the level of the abstract machine about the language. In contrast, for C and C++, we are forced to reason at the level of the concrete machine. To do this, we must understand what decisions the compiler writer makes, the details of the machine architecture, operating system, and other configuration details. In this sense, C and C++ aren’t *languages* because their meaning is not independent of their implementations.

As a simple example, consider a call `strcpy(x, 'abcdefghijklmnop')` where `x` is a buffer not big enough to hold the corresponding string. In practice, we know that this will result in overwriting the values in memory that are adjacent to the buffer `x`, but this is not necessary. Because such an operation is “undefined”, an implementor *could* translate this into the behavior that sends your credit card number to Bill Gates, or a behavior that causes an atomic bomb to be dropped on New York, or a behavior that causes cows to fly through the air. We cannot even begin to bound or write down the possible behaviors of such a program.

Of course, more likely, something next to `x` will simply be clobbered. But even to determine what actually *will* happen, we need to know what object(s) lie next to `x`, and this sort of detail is only available when we know all of the details of the compiler, run-time system, etc.

4.2 Static Typing

An alternative to dynamic typing (or punting the whole issue as in C), is to use static typing: We build a (decidable) classifier which determines whether or not an initial configuration will reach a stuck state. If

so, then we reject the term as not being a valid or well-formed program. Of course, for a Turing-complete language, if we could statically determine whether a program will become stuck, then we could solve the halting problem. So for such languages, static type-checking is conservative and will reject some programs that will not get stuck.

In practice, real languages use a mixture of dynamic and static checks. For instance, ML uses static checking to rule out most problems, but dynamic checks for match-incomplete, division-by-zero, and array-index-out-of-bounds.

We will specify the type-system for our language as a relation of the form $\Gamma \vdash e : \tau$ which you should read as “under the assumptions of Γ , the term e can be assigned the type τ ”. The context Γ is a list (or symbol table if you like) mapping variables to their types:

$$\text{(context)} \quad \Gamma ::= \bullet \mid \Gamma, x:\tau$$

It will be convenient (for now) to treat Γ as a partial function, and thus we will assume that there is at most one occurrence of each variable in a well-formed context, and that contexts are considered equivalent up to re-ordering. (Later, when we discuss sub-structural type systems, these assumptions will be thrown out.)

The typing rules are as follows:

$$\begin{aligned} \text{(const)} \quad & \Gamma \vdash c : b \\ \text{(var)} \quad & \Gamma \vdash x : \Gamma(x) \\ \text{(\(\rightarrow\ I\))} \quad & \frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \quad (x \notin \text{Dom}(\Gamma)) \\ \text{(\(\rightarrow\ E\))} \quad & \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \end{aligned}$$

Now our big claim is that if an expression can be assigned a type under an empty context (i.e., $\bullet \vdash e : \tau$), then that expression will not get stuck when it is evaluated. It’s pretty easy to see that stuck expressions won’t type-check under an empty context. But this is not enough — we must somehow rule out any expression that can step to a stuck state.

This can be achieved by proving two lemmas:

Preservation: if $\bullet \vdash e : \tau$ and $e \mapsto e'$, then $\bullet \vdash e' : \tau$.

Progress: if $\bullet \vdash e : \tau$, then either e is a value or else there exists an e' such that $e \mapsto e'$.

In essence, preservation tells us that “being well-typed” is a loop invariant for our small-step semantics. Progress tells us that the invariant is strong enough to establish that we won’t get stuck: rather, either we’ll reach a good terminal state or we can take a step. When you tie these two together, you see that there’s no way we can reach a stuck state.

This approach to proving type-soundness is called *subject reduction* and was popularized by Felleisen and Wright about a decade ago. It turns out to be a very straightforward and scalable technique (meaning we’ll be doing it over and over again.)

4.3 Homework

1. Using your favorite programming language, write an interpreter for the DeBruijn version of the lambda calculus. You’ll need to figure out how to adjust the indices (corresponding to free variables) on terms when you do substitution.
2. Prove preservation and progress for the CBN variant of the simply-typed lambda calculus.