
CMCS 312: Programming Languages
Lecture 6: Simply Typed Lambda Calculus

Umut A. Acar

October 12, 2006

Contents

1	Announcements	1
1.1	Quiz Solution	1
2	Introduction	3
3	Simply Typed Lambda Calculus	3
4	Types	4
4.1	Type Safety	6
4.2	Properties of Typing	7

1 Announcements

Your two homework exercises are due today, so please turn them in.

From now on, our late policy for the homeworks is 50% reduction for any late homeworks, unless you get permission from George ahead of time.

1.1 Quiz Solution

The problem was to give a call by value and call by name semantics for lambda calculus and show that both are deterministic.

Let's try to give a semantics for lambda calculus and see how the different evaluation strategies effect the decisions that we make.

As usual, let's start by writing out the syntax for lambda terms.

$$t ::= x \mid \lambda x. t \mid t_1 t_2$$

First, we need to determine what we consider a value. We will consider lambda abstractions as values, because both call-by-value and call-by-name evaluation strategies do so. Note that this is a choice. For example, the full beta-reduction strategy and the normal-order strategy do not consider the term $\lambda x. (\lambda x. x) x$ a value, because there is a redex inside the abstraction.

Now we will consider each term and give evaluation rules for them. We don't need an evaluation rule for variables, because we don't want to see a free (unbound) variable during evaluation—that would be an error.

Consider lambda abstractions. They are values. Since we are giving a small-step semantics, we need no rules for values.

Consider application, $t_1 t_2$. We know that we will apply beta reduction when t_1 is a lambda abstraction, *i.e.* $t_1 = \lambda x.t$. So we know what to do when t_1 is not value. Good, what if it is not a value? Then, we will simply evaluate it recursively.

How about t_2 ? Regardless of whether t_2 is a value or not, we can always use it in a beta reduction. So we have a choice to make here: should we wait until t_2 becomes a value to apply beta reduction, or should we go ahead with beta reduction? This is where we consult the evaluation strategy. The call-by-name strategy says, to go ahead and do the reduction. The call-by-value strategy says to wait until t_2 is a value.

Here is a possible call-by-value semantics.

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

$$\frac{t_2 \rightarrow t'_2}{t_1 t_2 \rightarrow t_1 t'_2}$$

$$\frac{}{(\lambda x.t)v \rightarrow [v/x] t}$$

Here is a possible call-by-name semantics.

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

$$\frac{}{(\lambda x.t)t_2 \rightarrow [t_2/x] t}$$

Is there anything strange with these? Note that the call-by-value semantics is non-deterministic, because when we have $t_1 t_2$, we can reduce any of them by applying any of the two rules. Since call-by-value semantics requires that both parts of an application be values before applying beta reduction, we need to figure which part we will reduce first. There is another choice to make here to determine the *evaluation order*. Typically, we reduce the left part of the application first. With this evaluation order, call-by-value corresponds to leftmost-innermost evaluation.

Here is one of the two possible deterministic call-by-value semantics.

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

$$\frac{t_2 \rightarrow t'_2}{(\lambda x.t_1) t_2 \rightarrow (\lambda x.t_1) t'_2}$$

$$\frac{}{(\lambda x.t)v \rightarrow [v/x] t}$$

It is now relatively straightforward to show that the semantics are indeed deterministic.

2 Introduction

So far, we have considered a few languages (e.g., a language for numbers, untyped lambda calculus) and gave an operational semantics for them. We observed several times that the operational semantics is not able to evaluate any term. In practice, this means that we would get “stuck” if we try to evaluate a term that the operational semantics does not know how to handle. An example of such a term is `if (λ x.x) then true else false`. This is why “unsafe” languages such as C/C++ can refuse to continue execution even after their compilers accept the program (e.g., “core dump”).

In this class, we will see how we can design languages that are “safe”, that is that never “go” wrong.”

3 Simply Typed Lambda Calculus

Values $v ::= \text{true} \mid \text{false} \mid \lambda x.t$
Terms $t ::= v \mid \text{if } t \text{ then } t \text{ else } t \mid t t$

Let’s give a call-by-value operational semantics for this

$$\frac{t_1 \rightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3}$$

$$\text{if true then } t_2 \text{ else } t_3 \rightarrow t_2$$

$$\text{if false then } t_2 \text{ else } t_3 \rightarrow t_3$$

$$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$$

$$\frac{t_2 \rightarrow t'_2}{t_1 t_2 \rightarrow t_1 t'_2}$$

$$\overline{(\lambda x.t)v \rightarrow [v/x] t}$$

Note now that this semantics gets stuck when we try to evaluate the term `if λx.x then true else false`.

4 Types

Think of all the terms that can be generated with the abstract syntax. The problem is that some of these terms does not make sense to our operational semantics. To address the problem, we will put some structure to the set of terms and divide into into subsets. These subsets, however, won't be arbitrary. We will instead group terms that the operational semantics treats similarly together into the same sets. We will then name each set with a *type*. The goal is to be able to say that if a term is *well typed*, the the operational semantics will be evaluate the term and give us a sensible value.

Given a term t , we will write $t : \tau$ to say that t has type τ . We can now define a typing relation for the simply typed lambda calculus more precisely using inference rules as follows. First, we will need to extend our syntax with a type definition.

Since our language has booleans, we have a type for booleans. Our language also has if statements, but we don't have a type for them. Because we want types to refer to values. The idea is that when we evaluate a term of type say τ , we want to get a value of type τ . In other words, types enable us to structure the terms of our language with respect to what kind of a result they will yield when evaluated.

Functions are also values, what should their type be? The type of a function should include its return type, because when we apply a function that is what it will evaluate to. How about the argument type? It is important to include that as well so that we can make sure that when a function is applied to an argument, the argument has the correct type.

Now here is a first try to a set of type inference judgements:

$$\frac{}{\text{true} : \text{bool}}$$

$$\frac{}{\text{false} : \text{bool}}$$

$$\frac{t_1 : \text{bool} \quad t_2 : \tau \quad t_3 : \tau}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau}$$

$$\frac{t : \tau_2}{\lambda x. t : \tau_1 \rightarrow \tau_2}$$

$$\frac{t_1 : \tau_1 \rightarrow \tau_2 \quad \tau_2 : \tau_1}{t_1 \ t_2 : \tau_2}$$

Note that the for the if statement we insist that the both branches have the same type. In other words, we insist that the both branches return values of the same type when evaluated. This is a conservative assumption, because we only need to know that the branch that ends up being evaluated returns the desired type. In particular, consider the following statement

`if true then true else (λ x.x)` always returns `true`. But it is not a well-typed term.

Question: This type system is broken. Why?

The problem is that the type system accepts very few terms. Consider for example the function `λx.if x then true else false`. We can assign a type to this term using the given judgments. In particular, if we apply the rules above, we will want to derive the relation `if x then true else false : bool` but this requires that we know the type of `x`. But we don't have a way of knowing the type of `x`.

To typecheck terms that have free variables in them, we will enrich our type system with a (*typing*) *context* for variables. We define a context for variables, written Γ as a function that maps variables to types. We define Γ as either empty, or a sequence of variable typings as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

We can now update our type system with the context.

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{\Gamma \vdash t_1 : \text{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$$

Note that we simply thread the context Γ through the rules, except that at function definitions. There is yet, one more thing that is not quite satisfactory with this type system. What is the type for `λx.x`? It can be any type really, e.g., it can be `bool → bool`, `(bool → bool) → (bool → bool)`. We always want types to be unique, i.e., we never want a term have more than one type. So we will extend our language so that we have type annotations at variable binding locations. Here is how our updated syntax and the type system looks like.

Types $\tau ::= \mathbf{bool} \mid \tau_1 \rightarrow \tau_2$
Values $v ::= \mathbf{true} \mid \mathbf{false} \mid \lambda x : \tau. t$
Terms $t ::= v \mid t t \mid \mathbf{if } t \mathbf{ then } t \mathbf{ else } t$
Context $\Gamma ::= \emptyset \mid \Gamma, x : \tau$

$$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash t_1 : \mathbf{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$$

4.1 Type Safety

We now have a set of typing rules for simply typed lambda calculus. But how do we know that these typing rules make sense? Recall that the motivation here was to design a language in which well-typed terms do not get stuck.

To prove that our design is sensible we will prove *type safety*. If we can indeed show that the type system is indeed safe, then we know that our typing rules are reasonable. How should we characterize type safety? What conditions do we need?

First remember that if we have a well typed term, then we want to be able to evaluate this term. Let's write this down more precisely:

If $t : \tau$, then either t is a value, or $t \rightarrow t'$.

This property says that if we are given well typed term, then the term is either a value (in this case we are done evaluating this term), or the term reduces to another term t' . This property is often called *progress*.

Note that progress property ties together types (the static semantics) and the operational semantics (dynamic semantics).

Does this suffice? May be, but by itself, this is not quite satisfactory, for example, you can have a semantics that evaluates a function to the constant

true. Such a semantics “obviously” does not make sense. Or rather, we will simply reject such semantics. Instead, we will insist that evaluation preserves typing. More precisely,

If $\Gamma \vdash t : \tau$ and $t \rightarrow t'$, then $\Gamma \vdash t' : \tau$. This property is often called *preservation*.

Let’s think about these properties together. They tell us that if we start with a program (a term) in our language, then either that term is a value, or the term can be reduced to another term without changing its type. The progress property is what makes types a crucial notion in programming. They establish types as an invariant that does not change through evaluation.

4.2 Properties of Typing

Before we prove type safety, let’s talk about some basic properties of typing and the structures that we use to state the type system. We will not prove these properties but I highly recommend that you do prove them yourself. The proofs should not be hard. These properties primarily give you some vocabulary for talking about type systems.

Lemma 1 (Inversion (of the typing relation))

If $\Gamma \vdash x : \tau$, then $x : \tau \in \Gamma$.

If $\Gamma \vdash \lambda x : \tau_1. t_2 : \tau$, then $\tau = \tau_1 \rightarrow \tau_2$ and $\Gamma, x : \tau_1 \vdash t_2 : \tau_2$.

If $\Gamma \vdash t_1 \tau_2 : \tau$, then there is some τ_1 such that $\Gamma \vdash t_1 : \tau_1 \rightarrow \tau$ and $\Gamma \vdash t_2 : \tau_1$.

If $\Gamma \vdash \mathbf{true} : \tau$, then $\tau = \mathit{Bool}$.

If $\Gamma \vdash \mathbf{false} : \tau$, then $\tau = \mathit{Bool}$.

If $\Gamma \vdash \mathbf{if } t_1 \mathbf{ then } t_2 \mathbf{ else } t_3 : \tau$ then $\Gamma \vdash t_1 : \mathit{bool}$, $\Gamma \vdash t_2 : \tau$ and $\Gamma \vdash t_3 : \tau$.

This lemma simply inverts the typing judgements. Since we define the type relation $t : \tau$ to be the minimum relation that is derivable by application of typing judgements, we can invert the type relation for each term.

Exercise: Show that given a context Γ , a term t has at most one type under Γ . (uniqueness of typing). Note that this holds for this type systems. It may not hold for others.

We now state what is known as a canonical forms lemma. This lemma states the forms of the values of each type.

Lemma 2 (Canonical Forms)

1. If v is a value and $v : \mathit{bool}$, then $v = \mathbf{true}$ or $v = \mathbf{false}$.

2. If v is value and $v : \tau_1 \rightarrow \tau_2$, then $v = \lambda x : \tau_2. t$.

Lemma 3 (Permutation)

If $\Gamma \vdash t : \tau$ and Δ is any permutation of Γ , then $\Delta \vdash t : \tau$.

Lemma 4 (Weakening)

If $\Gamma \vdash t : \tau$ and $x \notin \text{dom}(\Gamma)$, then $\Gamma, x : \tau' \vdash t : \tau$.