CMSC 336: Type Systems for Programming Languages

Lecture 5: Simply Typed Lambda Calculus

Acar & Ahmed **January 24, 2008**

Contents

1 Solution to the Exercise

1.1 Semantics for lambda calculus

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. For both call-by-value and call-by-name, lambda abstractions are values because no reductions are allowed inside them. Note that in full beta-reduction and in the normal-order strategy, we do not consider lambda abstractions as values, e.g., the term `λ x. (λ x. x) x` is not a value, because there is a redex inside the abstraction.

We now 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 a value, we will simply apply beta reduction. If t_1 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: we can apply beta reduction immediately or evaluate t_2 to a value first. This is where we consult the evaluation strategy. In call-by-name, we apply beta reduction immediately. In call by value we evaluate it.

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 $t_1\ t_2$, two rules apply. 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-outermost evaluation.

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

$$\frac{t_1 \;\rightarrow\; t_1'}{t_1\ t_2 \;\rightarrow\; t_1'\ t_2}$$

$$\frac{t_2 \;\rightarrow\; t_2'}{(\lambda x.t_1)\ t_2 \;\rightarrow\; (\lambda x.t_1)\ t_2'}$$

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

We can now show that semantics is deterministic. We need to prove that if $t \;\rightarrow\; t'$ and $t \;\rightarrow\; t''$ then $t' = t''$. I will do the proof for call by value (the case for call by name is symmetric). The proof is by induction on the depth of the derivation trees of the two reductions. When the depth of the derivation is 1, we know that we have an application. Since there is only one rule that we can possibly apply, the property holds. Suppose now that the property holds for all derivation trees of depth d or less. Consider now a derivation tree with depth $d + 1 > 1$. Consider now the last evaluation rules, if $t = t_1\ t_2$ and t_1 is a lambda abstraction, then by induction we know that t_2 will be evaluated deterministically and the lemma holds. If t_1 is not a lambda term, then there is only one rule to apply and the property holds by induction.

1.2 De Bruijn Indices

The de Bruijn notation of $tru\ :=\ \lambda x.\lambda y.x$ is $\lambda.\lambda.1$. The de Bruijn notation of $fls\ :=\ \lambda x.\lambda y.y$ is $\lambda.\lambda.0$. The de Bruijn notation for $pair\ :=\ \lambda x_1.\lambda x_2.\lambda y.\ y\ x_1\ x_2$ is $\lambda.\lambda.\lambda.0\ 2\ 1$.

For applying beta reduction in the de Bruijn notation, you needed to use the formula for substitution given in your notes. To define substitution on de Bruijn terms, we first need to define a shift operation that increases the indices of all free "variables" by a specified number. To know what variables are free, however, we need to know how deep we are in a term (in the abstract syntax tree). This is specified by a cutoff constant c. In the formula below, d is specifies by how much we want to shift the free variables.

$$\begin{aligned}
\uparrow_c^d (i) &= i \quad \text{if } i < c \\
\uparrow_c^d (i) &= i + d \quad \text{if } i \geq c \\
\uparrow_c^d (\lambda.t) &= \lambda.\uparrow_{c+1}^d (t) \\
\uparrow_c^d (t_1\ t_2) &= (\uparrow_c^d (t_1))\ (\uparrow_c^d (t_2))
\end{aligned}$$

We will write $\uparrow^i (t)$ to mean $\uparrow_0^i (t)$.

Examples:

1. $\uparrow^2 (\lambda.\lambda.1(0\ 2)) =$
2. $\uparrow^2 (\lambda.01(\lambda.0\ 1\ 2)) =$

We define substitution as follows. Shifting during substation is necessary to make sure that free variables of the substituted term still point to the same variables.

$$\begin{aligned}
[t/j]i &= t \ \text{ if } i = j \\
[t/j]i &= i \ \text{ if } i \neq j \\
[t/j]\lambda.t' &= \lambda.[\uparrow^1 (t)/j + 1]t' \\
[t/j]t_1\ t_2 &= [t/j]t_1\ [t/j]t_2
\end{aligned}$$

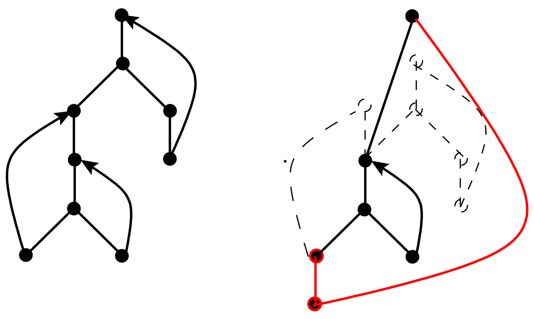

Figure 1: $\lambda.(\lambda.\lambda.1 0)\ (\lambda.1)\ \rightarrow_\beta\ \lambda.(\lambda.\lambda.2\ 0)$ illustrated

We define beta reduction by substituting the term for the 0th variables and shifting the term by -1, because beta reduction strips one lambda off the term.

$$(\lambda.t_1)t_2 = \uparrow^{-1} ([\uparrow^1 (t_2)/0]t_1)$$

2 Introduction

So far, we have considered a few languages (e.g., a language for numbers, untyped lambda calculus) and gave a 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`.

Unsafe languages such as C/C++ can refuse to continue execution when they encounter such terms (e.g., "core dump"). In this lecture, we will start learning some neat ideas for designing languages that don't get "stuck" or "go wrong."

3 Simply Typed Lambda Calculus

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

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

$$\frac{t_1 \;\rightarrow\; t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \;\rightarrow\; \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

$$\texttt{if true then } t_2 \texttt{ else } t_3 \;\rightarrow\; t_2$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \;\rightarrow\; t_3$$

$$\frac{t_1 \;\rightarrow\; t_1'}{t_1\ t_2 \;\rightarrow\; t_1'\ t_2}$$

$$\frac{t_2 \;\rightarrow\; t_2'}{t_1\ t_2 \;\rightarrow\; t_1\ t_2'}$$

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

Note now that this semantics gets stuck when we try to evaluate the term `if` $\lambda x.x$ `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 cannot be evaluated using our operational semantics. So we will try to identify the terms that the operational semantics can evaluate.

We will put some structure to the set of terms by dividing it into into disjoint subsets. Informally, we will place the terms that the operational semantics treats in the same way into the same subset. We will then give a name to each subsets which will be its *type*. The goal is to be able to say that if a term is *well typed*, then operational semantics will be able to evaluate the term to a value (barring infinite evaluation).

We have been vague about how we decide that the operational semantics treats two terms "similarly" (so that we can give them the same type). More precisely, we will assign two terms the same type if the operational semantics can evaluate them to a value of that type.

So we can build our type system by starting with the values, giving them a type and figuring out the types of other more complex terms. Since we have booleans in our language, we will have a type of `bool` type.

For a term t, we will write $t : \tau$ to say that t has type τ. So we have `true:bool` and `false:bool`.

Let's now start figuring out the type for other terms. We will do this inductively by using inference rules.

Before we go on, we will need to figure out a type for functions, but 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:

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

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

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

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

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

Note that for the if statement we insist that both branches have the same type. In other words, we insist that 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.

The reason for this assumption is that we want to be able to determine the type of the term by "looking" at it—we don't want to execute it. Execution would be too late. The idea here is to prevent execution time errors by designing a safe language. In other words, we want to be able to check a term and determine that it is safe to execute by simply inspecting it.

Question: This type system is broken. Why?

The problem is that it accepts very few terms. Consider for example the function $\lambda x.$`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.

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

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

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

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

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

Note that we simply thread the context Γ through the rules, except that at function definitions.

There is one more thing that is not quite satisfactory with this type system.

What is the type for $\lambda x.x$?

It can be any function type really, e.g., it can be $\texttt{bool} \ \rightarrow \ \texttt{bool}$, $(\texttt{bool} \ \rightarrow \ \texttt{bool}) \ \rightarrow \ (\texttt{bool} \ \rightarrow \ \texttt{bool})$.

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. Figure 2 shows our updated syntax and the type system.

$$
\begin{array}{llcl}
\textit{Types} & \tau & ::= & \texttt{bool} \mid \tau_1 \;\rightarrow\; \tau_2 \\[1em]
\textit{Values} & v & ::= & \texttt{true} \mid \texttt{false} \mid \lambda x : \tau . t \\[1em]
\textit{Terms} & t & ::= & v \mid t\ t \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\[1em]
\textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau
\end{array}
$$

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

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

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

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

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

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

Figure 2: Simply typed lambda calculus with booleans and its type system.

5 Type Safety

We now have a set of typing rules for simply typed lambda calculus. We do not know, however, that type system is faithful to the operational semantics. Recall that our motivation is to design a language where well typed terms do not get stuck during evaluation—when evaluating the term we never want to be in a situation where we don't know how to continue evaluating that term.

We will need to prove *type safety*.

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? Not quite. What about a semantics that evaluates a function to the constant `true`. 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 is often referred to as the *preservation* theorem.

Progress and preservation gives us some important invariants about about language. They tell us that if we start with a program (a closed term) in our language, then either that term is a value, or the term can be reduced to another term without changing its type.

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

If $\Gamma \vdash$ `true` $: \tau$*, then* $\tau = Bool$*.*

If $\Gamma \vdash$ `false` $: \tau$*, then* $\tau = Bool$*.*

If $\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \tau$ *then* $\Gamma \vdash t_1 : \texttt{bool}$, $\Gamma \vdash t_2 : \tau$ *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 $v : \texttt{bool}$, then $v = \texttt{true}$ or $v = \texttt{false}$.*
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* $x \notin \mathrm{dom}(\Gamma)$, *then* $\Gamma, x : \tau' \vdash t : \tau$.